Prabhat Mishra · Swarup Bhunia
Mark Tehranipoor *Editors*

Hardware IP Security and Trust

Springer

www.allitebooks.com

Hardware IP Security and Trust

www.allitebooks.com

Prabhat Mishra • Swarup Bhunia • Mark Tehranipoor
Editors

Hardware IP Security and Trust

Springer

www.allitebooks.com

Editors
Prabhat Mishra
Department of Computer and
Information Science and Engineering
University of Florida
Gainesville, FL, USA

Swarup Bhunia
Department of Electrical and
Computer Engineering
University of Florida
Gainesville, FL, USA

Mark Tehranipoor
Department of Electrical and
Computer Engineering
University of Florida
Gainesville, FL, USA

ISBN 978-3-319-49024-3 ISBN 978-3-319-49025-0 (eBook)
DOI 10.1007/978-3-319-49025-0

Library of Congress Control Number: 2016963250

© Springer International Publishing AG 2017
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

www.allitebooks.com

Acknowledgements

This book would not be possible without the contributions of many researchers and experts in the field of hardware security and trust. We would like to gratefully acknowledge the contributions from the following authors who have contributed book chapters:

- Adib Nahiyan, University of Florida, USA
- Debapriya Basu Roy, Indian Institute of Technology, Kharagpur, India
- Dr. Marc Stoettinger, TU Darmstadt, Germany
- Dr. Nicole Fern, University of California at Santa Barbara, USA
- Dr. Pramod Subramanyan, Princeton University, USA
- Farimah Farahmandi, University of Florida, USA
- Jaya Dofe, University of New Hampshire, USA
- Jonathan Frey, Draper, USA
- Kan Xiao, Intel, USA
- Prof. Debdeep Mukhopadhyay, Indian Institute of Technology, Kharagpur, India
- Prof. Domenic Forte, University of Florida, USA
- Prof. Hassan Salmani, Howard University, USA
- Prof. Qiang Xu, Chinese University of Hong Kong
- Prof. Qiaoyan Yu, University of New Hampshire, USA
- Prof. Sharad Malik, Princeton University, USA
- Prof. Sorin Alexander Huss, TU Darmstadt, Germany
- Prof. Tim Cheng, University of California at Santa Barbara, USA
- Prof. Yier Jin, University of Central Florida, USA
- Qihang Shi, University of Connecticut, USA
- Shivam Bhasin, Indian Institute of Technology, Kharagpur, India
- Sikhar Patranabis, Indian Institute of Technology, Kharagpur, India
- Yuanwen Huang, University of Florida, USA
- Yuejun Zhang, University of New Hampshire, USA

www.allitebooks.com

This work was partially supported by National Science Foundation (CNS-1441667, CNS-1558516, CNS-1603475, CNS-1603483), Semiconductor Research Corporation (2014-TS-2554, 2649) and Cisco. Any opinions, findings, conclusions or recommendations presented in this book are only those of the authors and contributors and do not necessarily reflect the views of the National Science Foundation, Semiconductor Research Corporation or Cisco.

www.allitebooks.com

Contents

www.allitebooks.com

www.allitebooks.com

Abbreviations (Acronyms)

3D	Three Dimensional
3PIP	Third-Party Intellectual Property
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
AMASIVE	Adaptable Modular Autonomous SIde-Channel Vulnerability Evaluator
ASIC	Application-Specific Integrated Circuit
ATPG	Automatic Test Pattern Generation
BDD	Binary Decision Diagram
BEOL	Back-End-of-Line
BFM	Bus Functional Model
BMC	Bounded Model Checking
CAD	Computer Aided Design
CAE	Computer-Aided Engineering
CBC	Cipher Block Chaining
CE	Circuit Editing
CEGIS	Counter-Example Guided Inductive Synthesis
CIC	Calculus of Inductive Construction
CISC	Complex Instruction Set Computer
CM	Central Moment
CNF	Conjunctive Normal Form
CPP	Cross-plane port
CS	Central Sum
CTL	Computation Tree Logic
DFD	Design-for-Debug
DFG	Data Flow Graph
DFT	Design for Test
DIP	Distinguishing inputs
DPA	Differential Power Analysis
DPM	Dynamic Power Management
DRC	Design Rule Check

www.allitebooks.com

DSD	Dynamic state-deflection
DSeRC	Design Security Rule Check
DSP	Digital Signal Processor
DTA	Dynamic Taint Analysis
DTM	Dynamic Thermal Management
ECC	Elliptic Curve Cryptography
ECP	Elliptic Curve Processor
EDA	Electronic design automation
EEPROM	Electrically Erasable Programmable Read-Only Memories
eMSK	enhanced Multi-Segment Karatsuba
EOFM	Electro-Optical Modulation
FANCI	Functional Analysis for Nearly-unused Circuit Identification
FF	Flip-flop
FIB	Focused Ion Beam
FIFO	First-In, First-Out
FIPS	Federal Information Processing Standard
FP	Floating Point
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FW	Firmware
HARPOON	Hardware protection through obfuscation of netlist
HDL	Hardware Description Language
HED	Horner Expansion Diagram
HiFV	Hierarchy-preserving Formal Verification
HLSM	High-Level State Machine
HT	Hardware Trojan
HW	Hardware
iATPG	Automatic Test Pattern Generation
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IFS	information flow security
IIR	Interrupt Identification Register
ILA	Instruction-Level Abstraction
Imp	Implementation
IoT	Internet of Things
IP	Intellectual Property
IR	Infra-Red
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
K-Map	Karnaugh Map
LED	Light-Emitting Diode
LFF	Largest Fanout First
LFSR	Linear Feedback Shift Register
LTL	Linear Temporal Logic
LUT	Lookup table

www.allitebooks.com

LVX	Laser Voltage Techniques
M1	Metal 1
MMIO	Memory Mapped Input/Output
MPU	Micro-Processor Unit
MRA	Mutating Runtime Architecture
MSB	Most Significant Bit
MSK	Multi-Segment Karatsuba
MUX	Multiplexer
NI	Network interface
NICV	Normalized Inter Class Variance
NoC	Network-on-chip
NOC-SIP	NoC shielding plane
OTP	One-Time Programmable Memory
OVM	Open Verification Methodology
PAD	Probe Attempt Detector
PBO	Pseudo Boolean Optimization
PC	Program Counter
PCC	Proof-Carrying Code
PCH	Proof-carrying hardware
PCHIP	Proof-carrying hardware IP
PE	Photon Emission
PoI	Point of Interest
POS	Product-of-Sums
PPIs	Pseudo Primary Inputs
PPOs	Pseudo Primary Outputs
PSL	Property Specification Language
PUF	Physical Unclonable Function
RAM	Random Access Memory
RDA	Received Data Available
RISC	Reduced Instruction Set Computer
ROBDD	Reduced Order Binary Decision Diagram
ROM	Read-only-memory
RSA	The Rivest-Shamir-Adleman cryptosystem
RSR	Reconfigurable ShiftRows
RTL	Register-Transfer Level
SAMD	Shared-Address Multiple-Data
SAT	Boolean Satisfiability Problem
SCA	Side Channel Analysis
SEM	Scanning electron microscopy
SHA-1	Secure Hash Algorithm 1
SM	Standardized Moment
SMT	Satisfiability Modulo Theories
SNR	Signal to Noise Ratio
SoC	System on Chip
SOP	Sum-of-Products

SPA	Simple Power Attack
Spec	Specification
SPP	The single-plane port
SRSWOR	Simple Random Sampling Without Replacement
SRSWR	Simple Random Sampling With Replacement
SSH	Secure Shell
STG	Signal Transition Graph
SW	Software
TLM	Transaction-level modeling
TLS	Transport Layer Security
TOCTOU	Time of Check To Time of Use
TSV	Through-Silicon-Via
TTP	trusted third party
TVLA	Test Vector Leakage Assessment
TVVF	Timing Violation Vulnerability Factor
UART	Universal Asynchronous Receiver/Transmitter
UCI	Unused Circuit Identification
UV	Ultra-Violet
VLIW	Very Large Instruction Word
XOR	Exclusive-Or

Part I
Introduction

Chapter 1
Security and Trust Vulnerabilities in Third-Party IPs

Prabhat Mishra, Mark Tehranipoor, and Swarup Bhunia

1.1 Introduction

Our daily life is intertwined in the fabric of Internet-of-Things (IoTs), a fabric in which the number of connected computing devices exceeds the human population. We anticipate over 50 billion devices to be deployed and mutually connected by 2020 [1]. Security and trust are paramount considerations while designing these systems. Majority of these IoT devices as well as embedded systems are designed using both System-on-Chip (SoC) components and software applications. In order to reduce design cost and meet shrinking time-to-market constraints, SoCs are designed using third-party Intellectual Property (IP) blocks [2]. For example, Fig. 1.1 shows a typical SoC with a processor core, a Digital Signal Processor (DSP), memory (MEM), analog-to-digital (ADC) and digital-to-analog (DAC) converters, communication network, peripherals (I/O), and so on. For the ease of illustration, communication network is shown as an IP block instead of connectivity between IPs and peripherals. In many cases, these IPs are designed by different companies across the globe and travel through long supply chain involving multiple vendors, as shown in Fig. 1.2. Security vulnerability of SoCs at different stages of its life cycle is an important concern to SoC designers, system integrators, as well as to the end users. Over the ages, hardware components, platforms, and supply chains have been considered secure and trustworthy. However, recent discoveries and reports on security vulnerabilities and attacks in microchips and circuits violate this assumption [3].

Growing reliance on reusable hardware IPs, often gathered from untrusted third-party vendors, severely affects the security and trustworthiness of SoC computing platforms [3]. An important emerging concern with the hardware IPs acquired from

P. Mishra (✉) • M. Tehranipoor • S. Bhunia
University of Florida, Gainesville, FL, USA
e-mail: prabhat@ufl.edu; tehranipoor@ece.ufl.edu; swarup@ece.ufl.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_1

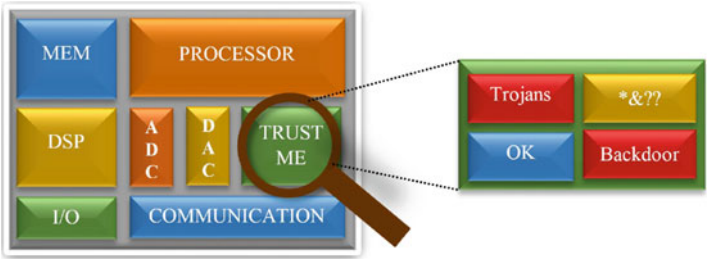


Fig. 1.1 Trusted SoC design using untrusted IPs

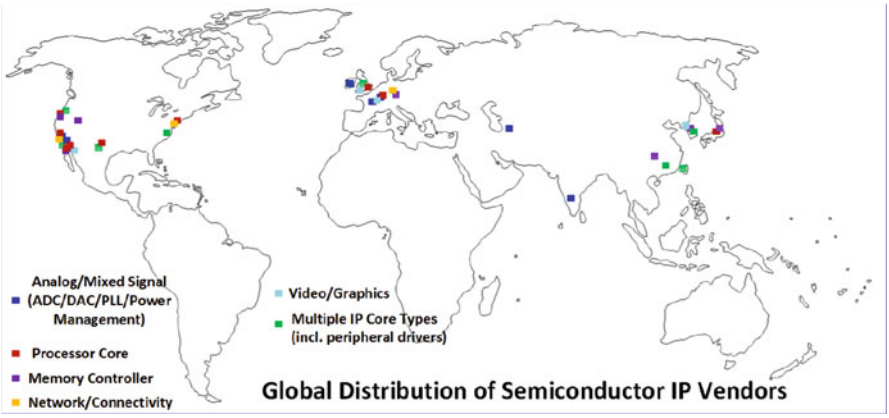


Fig. 1.2 The long and globally distributed supply chain for hardware IPs makes them increasingly vulnerable to diverse trust/integrity issues

external sources is that they may come with deliberate malicious implants to incorporate undesired functionality (e.g., hardware Trojan), undocumented test/debug interface working as a hidden backdoor, or other integrity issues. For example, a close inspection of an IP (listed as "TRUST ME" in Fig. 1.1) shows malicious logic (hardware Trojan), potential backdoor as well as unknown (unspecified) functionality. Untrusted IPs in an SoC can lead to security consequences for the end user. For example, a hardware Trojan in a processor module can leak the secure key or other protected data to the adversary. In fact, a hardware Trojan in the memory IP would be able to access the secure key from the processor IP if the attacker can carefully analyze the state transitions and able to reach a protected state in the processor IP from any state in the memory IP. It is important to note that an IP from a trusted company may have malicious implants if the IP was outsourced during any of the design stages such as synthesis, verification, test insertion (DFT), layout, fabrication, manufacturing testing, and post-silicon debug. Depending on the type of IP vulnerability, an adversary can not only exploit hardware Trojans, but can also perform side-channel attack, probing attack, FSM vulnerability-based attack, and

so on. This book provides a comprehensive coverage of SoC vulnerability analysis and presents effective countermeasures and validation techniques to design trusted SoCs using untrusted third-party IPs.

This chapter is organized as follows. Section 1.2 describes the current practice of SoC design and validation methodology. Section 1.3 outlines security and trust vulnerabilities in third-party IPs. Section 1.4 presents how to design trustworthy SoCs using potentially untrusted third-party IPs. Finally, Sect. 1.5 provides the organization of the remaining chapters.

1.2 Design and Validation of SoCs

System-on-Chip (SoC) integrates both hardware and software components into an Integrated Circuit (IC) that can perform the desired functionality. The hardware components can be processors, co-processors, controllers, converters (analog-to-digital and digital-to-analog), FPGA, memory modules, and peripherals. Similarly, the software components can be real-time operating systems and control programs. A typical SoC can perform digital, analog as well as mixed-signal computations. Figure 1.1 shows a block diagram of a typical SoC design. The example SoC consists of processor, DSP, memory, converters, communication network, and peripherals. SoCs are widely used in designing both IoT devices and embedded systems in a wide variety of domains including communication, entertainment, multimedia, automotive, avionics, and so on. The remainder of this section outlines SoC design methodology followed by SoC validation techniques.

Figure 1.3 shows a typical top-down SoC design methodology. It starts with a design specification, followed by hardware–software co-design (includes partitioning, code generation for software models, and synthesis, layout, and fabrication for hardware models), and integration and verification. Some semiconductor companies use the platform-based SoC design, which is a variation of the traditional top-down methodology. In platform-based methodology, desired behavior (specification) is mapped to the target platforms. Some parts of the design behavior may be available as IPs, whereas the remaining parts need to be implemented in hardware or software. Due to increasing SoC design complexity coupled with reduced time-to-market constraints, semiconductor companies have opted for IP-based SoC design methodology. This enables them to put together all the required components very quickly (compared to top-down design methodology) followed by integration and verification as outlined in Fig. 1.4.

SoC validation is a major challenge due to combined effects of increasing design complexity and reduced time-to-market. Recent studies report that SoC validation consumes up to 70 % resources (cost and time) of overall SoC design methodology [4]. This cost computation includes system-level validation of architectural models, pre-silicon validation as well as post-silicon validation of both functional behaviors and non-functional requirements. Simulation is widely used form of SoC validation using random, constrained random as well as directed test vectors. Since the total

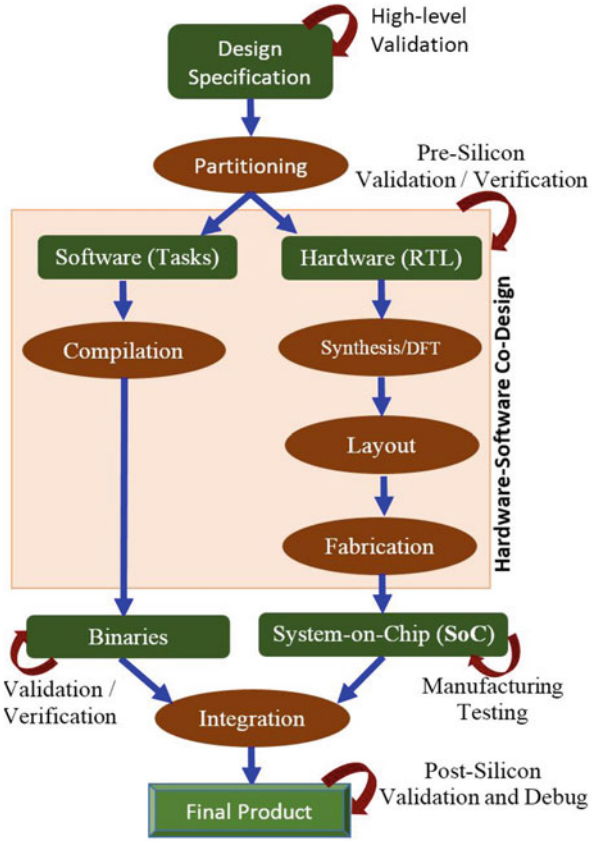


Fig. 1.3 Traditional top-down SoC design methodology

number of possible input combinations (test vectors) can be exponential, simulation-based approaches cannot guarantee completeness of verification. On the other hand, formal methods, such as model (property) checking, equivalence checking, and satisfiability (SAT) solving, can prove correctness but cannot handle large designs due to state-space explosion. SoC verification experts use an efficient combination of simulation-based techniques and formal methods. In a typical industrial setting, full SoC is simulated using efficient test vectors, whereas some of the critical components (e.g., security IPs and complex protocols) are verified using formal methods. Unfortunately, these validation techniques are not designed to detect security and trust vulnerabilities of SoCs [5–8].

1.3 Security and Trust Vulnerabilities in Third-Party IPs

Hardware IPs acquired from untrusted third-party vendors can have diverse security and integrity issues. An adversary inside an IP design house involved in the IP design process can deliberately insert a malicious implant or design modification

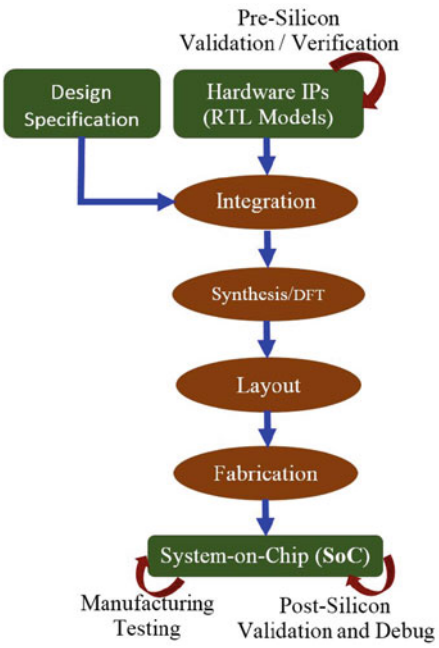


Fig. 1.4 IP-based SoC design methodology

to incorporate hidden and undesired functionality. Such additional functionalities can serve broadly two purposes for an adversary: (1) it can cause malfunction in the SoC that integrates the IP; and (2) it can facilitate information leakage through a hardware backdoor that enables unauthorized access or can directly leak secret information (e.g., cryptographic key, or internal design details of an SoC) from inside a chip. Wide array of hardware Trojan designs including different forms of combinational and sequential Trojans investigated by researchers, with payloads causing malfunction or information leakage, would form a possible arsenal for an adversary. An information leakage or hidden backdoor can give an attacker illegal access to important systems or valuable data inside SoC during in-field deployment of a system. Such an access can potentially be gained remotely through a hardware backdoor. On the other hand, sudden malfunction of an electronic component triggered by a malicious modification can allow an attacker to cause catastrophic consequences in critical infrastructure. Both types of effects constitute critical concerns with respect to national and personal security.

In addition to deliberate malicious changes in a design, IP vendors can also unintentionally incorporate design features, e.g., hidden test/debug interfaces that can create critical security loopholes. In 2012, a breakthrough study by a group of researchers in Cambridge revealed an undocumented silicon level backdoor in a highly secure military-grade ProAsic3 FPGA device from MicroSemi (formerly Actel) [9]. Similarly, IPs can have uncharacterized parametric behavior (e.g., power/thermal) which can be exploited by an attacker to cause irrecoverable damage to an electronic system. In a recent report, researchers have demonstrated such

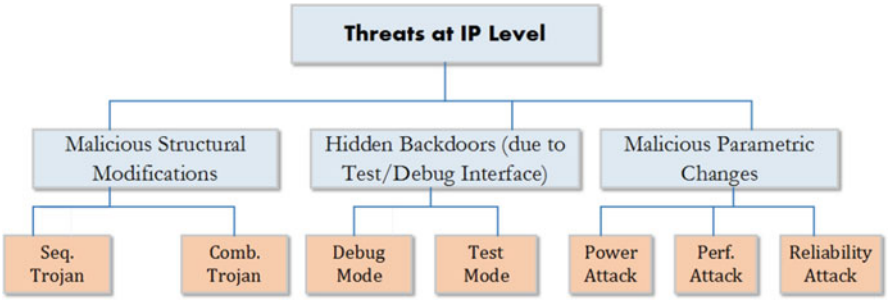


Fig. 1.5 The taxonomy of trust issues for hardware IPs

an attack where a malicious upgrade of a firmware destroys the processor it is controlling by affecting the power management system [10]. It manifests a new attack mode for IPs, where firmware/software update can maliciously affect the power/performance/temperature profile of a chip to either destroy a system or reveal secret information through appropriate side-channel attack, e.g., a fault or timing attack [11]. Hence, there is a critical need to perform rigorous and comprehensive analysis of trust/integrity issues in hardware IPs and develop effective low-cost countermeasures to protect SoC designers as well as end users against these issues.

Digital IP comes in several different forms: (1) soft IP (e.g., register transfer level IP), (2) firm IP (e.g., gate level IP), and (3) hard IP (e.g., layout level IP). All forms of IPs are vulnerable to the above-mentioned trust issues. Figure 1.5 shows the major classes of trust/integrity issues for hardware IPs. First, an adversary can implant combinational and sequential Trojans, which are triggered by internal node conditions. It is expected that an intelligent adversary would make the trigger condition of these Trojans rare, to evade detection during regular functional testing [12]. A combinational Trojan is activated by a trigger condition, which is a Boolean function of internal node values. On the other hand, a sequential Trojan is activated by a sequence of rare conditions and is realized with a state machine comprising of one or more state elements. In general, these Trojans can have two types of payloads (which indicate the effect of Trojan): either malfunction or information leakage, as mentioned earlier. The latter can be accomplished by leaking secret information either through output ports as logic values or through side-channel information leakage, i.e., adding single bit of information at a time to the power signature to leak the key from a crypto IP [13]. Both types of Trojans can have varying complexity, which are expected to correlate well with the triggering probability. A more complex combinational (or sequential) Trojan is likely to use more nodes (trigger inputs) to realize a trigger condition and hence a more rare activation condition.

The second class of trust/integrity issues would relate the apparently benign test/debug interface which makes IPs vulnerable to diverse security attacks during field operation. For example, a scan chain in crypto IP that interfaces with key-dependent logic, if accessible postfabrication in an SoC, can enable mounting a

scan-based controllability/observability attack leading to leakage of the secret key. Similarly, debug interface in an SoC that allows post-silicon validation, debug, and failure analysis is well-known vulnerability for diverse attacks. A recent incident points to such a vulnerability [9]. A study revealed an undocumented silicon level backdoor in a highly secure military-grade reconfigurable device from MicroSemi. This backdoor had a key or passcode as Microsemi puts it, which was extracted in affordable time and effort by the researchers using a highly sensitive differential power analysis technique. With this backdoor, an attacker has access to different chip components like non-volatile memory (NVM), array bitstream, etc. An attacker can utilize this to remotely reprogram as well as disable the chip even if a user locks it with his/her own user key. Microsemi has confirmed the presence of this structure as a test/debug interface with JTAG, controllable, and usable only by the manufacturer. The concern is that they cannot be fixed by software antivirus and hence those chips which are deployed remain susceptible. It is important to analyze if such a test/debug interface creates either direct (logic-based) or indirect (side-channel) vulnerability in terms of information leakage.

The third class of Trojan relates to hidden input/operating conditions that can bring an IP outside safe operating points, such as attacks where an IP violates the peak power or temperature constraint. Since many third-party IPs may not be characterized well with their parametric behavior, it makes them vulnerable with respect to attacks where an attacker tries to exercise input (possibly rare) conditions to violate non-functional constraints. Such a condition can impose serious threat where the firmware/software/input can be used to directly attack the hardware, as reported in [10]. This book presents analysis of major types of attacks at the IP level and their implications in SoC security and trust.

1.4 Trustworthy SoC Design Using Untrusted IPs

Recent research efforts have targeted IP trust assurance. These efforts can be classified into two broad categories: (1) test and verification solutions, where a set of test patterns aim at identifying unknown hidden Trojans in an IP; and (2) design solutions, where SoC designers follow a pre-defined protocol with the IP providers. Existing works on test/verification solutions have considered both statistical and directed test generation. The scalability of the approach to large designs or various forms of IPs and Trojans has not been extensively studied. A salient work in the second category is based on embedding proof-carrying-code (PCC) into a design, where the SoC designer provides a set of proofs to an IP vendor for embedding into the IP, which can then be checked by the SoC designer for integrity verification. While such a technique can be effective to provide high level of assurance to SoC designer on certain properties and components of an IP, it cannot provide comprehensive assurance in all structural components of an IP. An attacker of the IP can evade the protection of PCCs by incorporating a Trojan in logic blocks not covered by them. Furthermore, it may incorporate considerable design overhead due

to integration of the PCCs and imposes a significant requirement and binding for the SoC designer to interact with an IP vendor during the IP design process.

Due to their stealthy nature and practically infinite Trojan space (in terms of trigger conditions, forms, and sizes), a cleverly inserted Trojan in an IP integrated into a large SoC is highly likely to evade conventional post-silicon testing. Furthermore, malicious hardware can easily bypass traditional software-implemented defense techniques as it is a layer below the entire software stack. Similarly, an apparently benign debug/test interface or uncharacterized parametric behavior under rare input conditions can be exploited by an adversary during field usage. Existing security analysis and trust validation at IC level do not readily extend to hardware IPs used in an SoC due to the following reasons.

- Different Threat Model: Trust/integrity issues at IP level fall into a fundamentally different class than those at IC primarily because third-party IPs do not come with golden reference models. It prevents performing conventional sequential equivalence check (SEC) to identify an anomaly. Most often these IPs come with simply a functional specification (often incomplete) and optionally high-level block diagrams. On the other hand, for IC, either a set of golden ICs or a golden design at some abstraction level is used as a reference.
- Hidden Test/Debug Interface: Unlike ICs, where test/debug interfaces that go into a design are available to test engineers during trust validation, IPs can include hidden test/debug interface that can create major security issue, as discovered recently.
- Parametric Vulnerability: Power/temperature/performance behavior of an IP may not be adequately verified by an IP vendor, which can create unanticipated vulnerability, as mentioned earlier.

Fortunately, an IP of any form provides an important advantage with respect to trust verification. It is not really a black box and one can exploit the structural/functional information about the constituent logic blocks (e.g., datapath) to efficiently identify diverse trust issues. Three major types of IP trust validation solutions, as illustrated in Fig. 1.6, are detailed below.

Functional Analysis The functional analysis can be performed in two steps. First, it can employ a statistical random testing approach, which aims at activating arbitrary Trojan instances. Published results in this area show that such an approach can be highly effective for small Trojans, particularly combinational ones [14]. It tries to bias a random test generator toward possible malicious implants. However, such an approach generally is not effective for larger Trojans whose activation conditions are random-resistant. Statistical random tests will be complemented with directed tests.

Structural Analysis It is important to ensure that an IP block performs exactly as intended in the specification (nothing more, nothing less), which requires evaluation of both state machine and datapath. To check the integrity of the state transition function, one can employ a self-referencing approach [15], which compares the temporal behavior of a finite state machine (FSM) at two different time instants

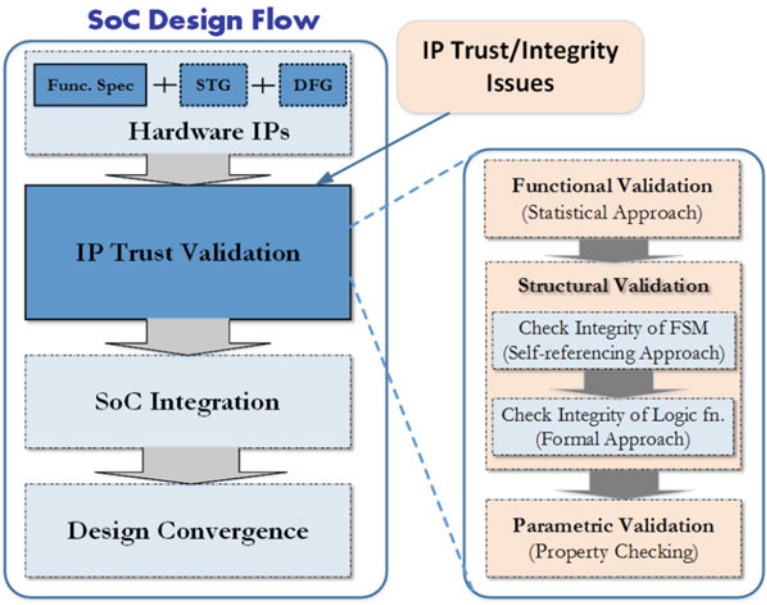


Fig. 1.6 Different classes of IP trust validation approaches which can potentially be integrated for higher level of confidence

to identify malicious modification of the FSM. The basic idea is that a sequential Trojan (e.g., a counter Trojan) in a design would trigger uncorrelated activities in the state machine and hence can be isolated. Self-referencing will be complemented with a formal verification approach. Formal verification methods make use of mathematical techniques to guarantee the correctness of a design with respect to some specified behavior. Unfortunately, existing methods often fail for large IP blocks due to state-space explosion during equivalence checking unless the structure of specification and implementation are very similar. Hence, new scalable formal methods for structural trust verification of IP need to be developed [7].

Parametric Analysis It is also important to check various properties of IPs including peak power and peak temperature for trust assurance. An attacker can generate a power virus (through software/firmware/input control) that can maximize the switching activity in the IP. In case the peak power violates the threshold, the designers may need to re-design the IP block to reduce the peak power. An attacker can develop mechanisms to produce a temperature virus that can generate sustained peak power to produce peak temperature. In case the temperature virus can increase the IP temperature beyond threshold, it can lead to reliability, security, and safety concerns.

Secure by Design As an alternative or complementary approach to IP level trust validation, judicious design solutions can be used to protect SoCs from malicious

behavior of untrusted IPs. Such a solution would often require collaboration between SOC design house and IP vendors. The central idea for such secure by design solution is to design an IP with specific provable properties whose trustworthiness can be easily established during SoC integration. An SoC designer would ask an IP vendor to embed a set of pre-defined verifiable security assertions into an IP which can be checked—e.g., using formal methods, to verify the trust level of the IP. Any violation of the security assertions/checks would indicate the IP being non-compliant and hence potentially malicious. Major challenges with this approach include derivation of appropriate security checks for different functional blocks in a design, the design overhead due to insertions of security checks, and finally, difficulty to safe-guard the entire design. Besides, such an approach requires change in the current business model where IPs are sold to chip manufacturers as commodity products with little or no customization.

1.5 Book Organization

This book is organized as follows. Chapters 2–5 present several trust vulnerability analysis techniques. Chapters 6–7 provide effective countermeasures against various forms of attacks. Chapters 8–14 survey efficient techniques for testing and validation of hardware IP security and trust. Finally, Chap. 15 concludes the book.

- *Chapter 2* presents a framework to analyze design vulnerabilities at different abstraction levels and assess its security at design stage.
- *Chapter 3* describes vulnerability analysis for both gate- and layout-level designs to quantitatively determine their susceptibility to hardware Trojan insertion.
- *Chapter 4* presents an interesting case study to identify suspicious signals using both formal and semi-formal coverage analysis methods.
- *Chapter 5* surveys existing techniques in performing probing attacks, the problem of protection against probing attacks, and presented a layout-driven framework to assess designs for vulnerabilities to probing attacks.
- *Chapter 6* reviews three major security hardening approaches—camouflaging, logic encryption/locking, and design obfuscation—that are applied to ICs at layout, gate, and register transfer levels.
- *Chapter 7* presents a mutating runtime architecture to support system designers in implementing cryptographic devices hardened against side-channel attacks.
- *Chapter 8* surveys the existing security validation methods for soft IP cores using a combination of simulation-based validation and formal methods.
- *Chapter 9* utilizes model checking and theorem proving using proof-carrying hardware for trust evaluation.
- *Chapter 10* describes three methods to detect potential hardware Trojans by utilizing Trojan characteristics. It also outlined how a stealthy Trojan can evade these methods.

- *Chapter 11* outlines how to exploit unspecified functionality for information leakage and presented a framework for preventing such attacks.
- *Chapter 12* proposes mechanism for specifying security properties and verifying these properties across firmware and hardware using instruction-level abstraction.
- *Chapter 13* describes how to perform test generation for detecting malicious variations in parametric constraints.
- *Chapter 14* covers side-channel testing using three metrics, along with practical case studies on real unprotected and protected targets.
- *Chapter 15* concludes the book and provides a future roadmap for trustworthy SoC design.

References

1. D. Evans, Cisco white paper on the internet of things: how the next evolution of the internet is changing everything (April 2011). www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf. Accessed 8 Nov 2016
2. M. Keating, P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs* (Kluwer, Norwell, MA, 1998)
3. S. Bhunia, D. Agrawal, L. Nazhandali, Guest editors' introduction: trusted system-on-chip with untrusted components. IEEE Des. Test Comput. **30**(2), 5–7 (2013)
4. M. Chen, X. Qin, H. Koo, P. Mishra, *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques* (Springer, New York, 2012)
5. Y. Huang, S. Bhunia, P. Mishra, MERS: statistical test generation for side-channel analysis based Trojan detection, in *ACM Conference on Computer and Communications Security (CCS)* (2016)
6. X. Guo, R. Dutta, P. Mishra, Y. Jin, Scalable SoC trust verification using integrated theorem proving and model checking, in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2016), pp. 124–129
7. F. Farahmandi, Y. Huang, P. Mishra, Trojan localization using symbolic algebra, in *Asia and South Pacific Design Automation Conference (ASPDAC)* (2017)
8. X. Guo, R. Dutta, Y. Jin, F. Farahmandi, P. Mishra, Pre-silicon security verification and validation: a formal perspective, in *ACM/IEEE Design Automation Conference (DAC)* (2015)
9. S. Skorobogatov, C. Woods, Breakthrough silicon scanning discovers backdoor in military chip, in *CHES 2012*. Published in Lecture Notes in Computer Science, vol. 7428 (2012), pp. 23–40
10. Ellen Messmer, RSA security attack demo deep-fries Apple Mac components, Network World (2014). http://www.networkworld.com/article/2174737/security/rsa-security-attack-demo-deep-fries-apple-mac-components.html
11. P.C. Kocher, Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems, in *CRYPTO* (1996), pp. 104–113
12. F. Wolff, C. Papachristou, R. Chakraborty, S. Bhunia, Towards Trojan-free trusted ICs: problem analysis and a low-overhead detection scheme, in *Design Automation and Test in Europe (DATE)* (2008)
13. L. Lin, W. Burleson, C. Parr, MOLES: malicious off-chip leakage enabled by side-channels, in *International Conference on CAD (ICCAD)* (2009)

14. R. Chakraborty, F. Wolff, S. Paul, C. Papachristou, S. Bhunia, MERO: a statistical approach for hardware Trojan detection, in *Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (2009)
15. S. Narasimhan, X. Wang, D. Du, R. Chakraborty, S. Bhunia, TeSR: a robust temporal self-referencing approach for hardware Trojan detection, in *4th IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2011)

Part II
Trust Analysis

Chapter 2
Security Rule Check

Adib Nahiyan, Kan Xiao, Domenic Forte, and Mark Tehranipoor

2.1 Introduction

With the emergence of information technology and its critical role in our daily lives, the risk of cyber attacks is larger than ever before. Many security systems or devices have critical assurance requirement. Their failure may endanger human life and environment, cause serious damage to critical infrastructure, hinder personal privacy, and undermine the viability of whole business sectors. Even the perception that a system is more vulnerable than it really is (e.g., paying with a credit card over the Internet) can significantly impede economic development. The defense against intrusion and unauthorized use of resources with software has gained significant attention in the past. Security technologies, including antivirus, firewall, virtualization, cryptographic software, and security protocols, have been developed to make systems more secure.

While the battle between software developers and hackers has raged since the 1980s, the underlying hardware was generally considered safe and secure. However, in the last decade or so, the battlefield has expanded to hardware domain, since emerging attacks on hardware are shown to be more effective and efficient than traditional software attacks in some aspects. For example, while the cryptographic algorithms have been improved and become extremely difficult (if not impossible) to break mathematically, their implementations are often not. It has been demonstrated that the security of cryptosystems, system on chips (SoCs), and microprocessor circuits can be compromised using timing analysis attacks [1], power analysis

A. Nahiyan (✉) • D. Forte • M. Tehranipoor
University of Florida, Gainesville, FL, USA
e-mail: adib1991@ufl.edu; dforte@ece.ufl.edu; tehranipoor@ece.ufl.edu

K. Xiao
Intel, Santa Clara, CA, USA
e-mail: kan.xiao@intel.com

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_2

attacks [2], exploitation of design-for-test (DFT) structures [3–5], and fault injection attacks [6]. These attacks can effectively bypass the security mechanisms built in the software level and put devices or systems at risk. These hardware based attacks aim to exploit the vulnerabilities in the design which are introduced either unintentionally or intentionally during the IC design flow.

Many security vulnerabilities in ICs can be unintentionally created by design mistakes and designer's lack of understanding of security problems. Further, today's CAD tools are not equipped with understanding security vulnerabilities in integrated circuits. Therefore, a tool can introduce additional vulnerabilities in the circuit [7, 8]. These vulnerabilities can facilitate attacks such as fault injection or side-channel based attacks. Also, these vulnerabilities can cause sensitive information to be leaked through observable points which are accessible to an attacker or give unauthorized access to an attacker to control or affect a secured system.

Vulnerabilities can also be intentionally introduced in ICs in the form of malicious modifications, referred to as hardware Trojans [9]. Due to short time-to-market constraints, design houses are increasing being dependent on third party to procure IPs. Also, due to the ever increasing cost of manufacturing ICs, design houses rely on untrusted foundry and assembly for fabricating, testing, and packaging ICs. These untrusted third party IP owners or foundries can insert hardware Trojans to create backdoors in the design through which sensitive information can be leaked and other possible attacks (e.g., denial of service, reduction in reliability, etc.) can be performed.

It is of paramount importance to identify security vulnerabilities during hardware design and validation process, and address them as early as possible due to the following reasons: (1) there is little or no flexibility in changing or updating post-fabricated integrated circuits; (2) The cost of fixing a vulnerability found at later stages during the design and fabrication processes is significantly higher following the well-known rule-of-ten (the cost of detecting a faulty IC increases by an order of magnitude as we advances through each stage of design flow). Moreover, if a vulnerability is discovered after manufacturing while the IC is in-field, it may cost a company millions of dollars in lost revenues and replacement costs.

Identifying security vulnerabilities requires extensive knowledge of hardware security, which design engineers lack due to the high complexity and diversity of hardware security issues. Hence, hardware security engineers are required to analyze circuit implementations and specification, and identify potential vulnerabilities. This requires engineers with significant knowledge of different vulnerabilities stemming from diverse set of existing and emerging attacks. It is prohibitively expensive for a design house to maintain a large team of security experts with high expertise while the growing complexity of modern designs significantly increases the difficulty of manual analysis of security vulnerabilities. Poor security check could result in unresolved security vulnerabilities along with large design overhead, development time, and silicon cost [10].

Such limitations suggest to automate the process of security vulnerability analysis during design and validation phases. In this chapter, we present a framework, called Design Security Rule Check (DSeRC) [11], to be integrated in the

conventional design flow to analyze vulnerabilities of a design and assess its security at various stages of design process, namely register transfer level (RTL), gate-level netlist, design-for-test (DFT) insertion, physical design, etc. DSeRC framework is intended to be used as security subject matter expert for design engineers. To achieve this, one needs a comprehensive list of vulnerabilities for ICs. The vulnerabilities are then tied with rules and metrics, so that each vulnerability can be quantitatively measured. The DSeRC framework will allow the semiconductor industry to systematically identify vulnerabilities and security issues before tape-out in order to include proper countermeasures or refine the design to address them.

The rest of the chapter is organized as follows: In Sect. 2.2, we present what are the security assets and attack models. Here, we also talk about who are potential adversaries and how they can get unauthorized access to assets. In Sect. 2.3, we present the proposed DSeRC framework, vulnerabilities, and the associated metrics rules. We present in detail how the vulnerabilities are introduced at different stages of design process. We also present a brief overview of the rules and metrics which are required to quantitatively analyze vulnerabilities. In Sect. 2.4, we discuss the required tasks for the development DSeRC framework. Finally, Sect. 2.5 concludes the chapter.

2.2 Security Assets and Attack Models

To build a secure integrated circuit, a designer must decide what assets to protect, and which of the possible attacks to investigate for. Further, IC designers must also understand who the players (attackers and defenders) are in the IC design supply chain and have the means for quickly evaluating the security vulnerabilities and the quality of the countermeasures against a set of well-defined rules and metrics. Three fundamental security factors (security assets, potential adversaries, and potential attacks) are associated with security checks in integrated circuits, which are discussed in the following.

2.2.1 Asset

As defined in [10], asset is a resource of value which is worth protecting with respect to the adversary. An asset may be a tangible object, such as a signal in a circuit design, or may be an intangible asset, such as controllability of a signal. Sample assets that must be protected in an SoC are listed below [12]:

- **On-device key:** Secret key, e.g., private key of an encryption algorithm. These assets are stored on chip in some form of non-volatile memory. If these are breached, then the confidentiality requirement of the device will be compromised.

- **Manufacture firmware:** Low level program instructions, proprietary firmware. These assets have intellectual property values to the original manufactures and compromising these assets would allow an attacker to counterfeit the device.
- **On-device protected data:** Sensitive data such as user's personal information and meter reading. An attacker can invade someone's privacy by stealing these assets or can benefit himself/ herself by tampering with these assets (meter reading).
- **Device configuration:** Configuration data determining which resources are available to users. These assets determine which particular services or resources are available to a particular user and an attacker may want to tamper with these assets to gain illegal access to these resources.
- **Entropy:** Random numbers generated for cryptographic primitives, e.g., initializing vector or cryptographic key generation. Successful attacks on these assets would weaken cryptographic strength of a device.

The security assets are known to the hardware designers based on the target specifications of a design. For example, a designer knows that the private encryption key used by the crypto-module is an asset and also knows where the key is located in the SoC. Different types of assets and their locations in an SoC are shown in Fig. 2.1.

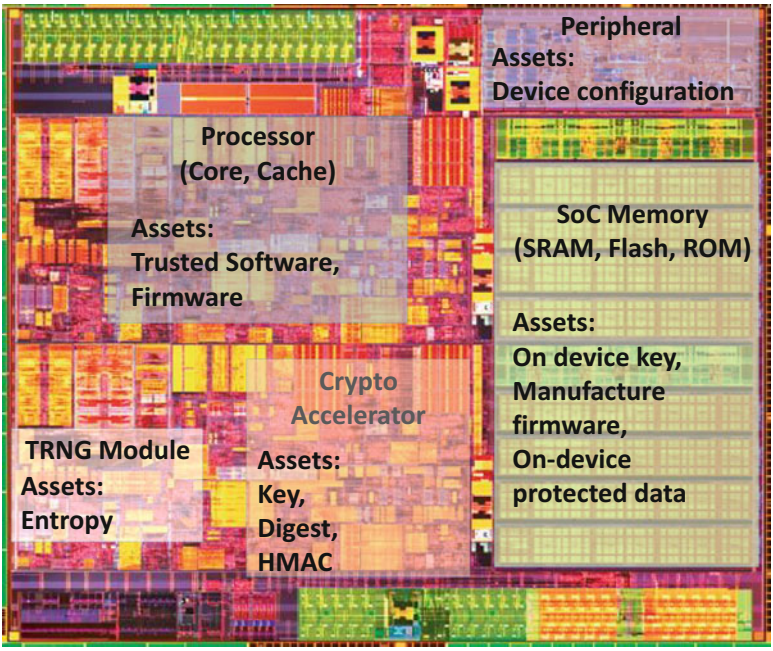


Fig. 2.1 Assets in SoC. *Source: Intel*

www.allitebooks.com

2.2.2 Potential Access to Assets

The aim of an attack is usually to obtain access to assets. There are three types of attacks to gain access to assets depending on attackers' abilities: remote attacks, non-invasive physical attacks, and invasive physical attacks.

Remote Attacks In this case an attacker has no physical access to the device; i.e., the attacker cannot touch the device. The attacker can still perform timing [13] and electromagnetic [14] side-channel attacks to remotely extract private key from devices such as smartcards. It has also been demonstrated that an attacker can remotely access the JTAG port and compromise the secret key stored in smartcard of a set-top box [15].

It is also possible to remotely access the scan structure of a chip. For example, in automotive applications, the SoC controlling critical functions such as breaks, power-train, air bags go into "test-mode" every time the car is turned off or on. This key-off/on tests ensure that these critical systems are tested and working correctly before every drive. However, modern cars can be remotely turned on or off, either by trusted parties such as roadside assistance operators or by malicious parties as shown in recent news. Remotely turning the car on or off, allows access to the SoC's test mode which can be used to obtain information from the on-chip memory or force unwanted functions.

Remote attacks also include those that utilize the weakness of hardware, such as buffer overflow, integer overflow, heap corruption, format string, and globbing [16].

Non-Invasive Physical Attacks Such attacks are usually of low-budget and do not cause the destruction of the device under attack. Basic attacks consist of using the primary inputs and outputs to take advantage of security weaknesses in the design to obtain sensitive information. Additionally, more advanced attacks use JTAG debug, boundary scan I/O, and DFT structures to monitor and/or control system intermediate states, or snoop bus lines and system signals [4]. Other attacks consist of injecting faults to cause an error during computation of cipher algorithms and exploit the faulty results to extract the asset (e.g., private key).

Semi-Invasive Physical Attacks Semi-invasive attacks fall between non-invasive and invasive physical attacks. These attacks present a greater threat because they are more effective than non-invasive attacks but can be performed at a much lower cost than an invasive physical attack. Semi-invasive physical attacks require partial depackaging the chip to get access to its surface; but unlike invasive attacks, these attacks do not require complete removal of the internal layers of the chip. Such attacks include injecting faults to modify SRAM cells content or change the state of a CMOS transistor and gain control of a chip's operation or bypass its protection mechanisms [17].

Invasive Physical Attacks These fall under the most sophisticated and expensive attacks and require advanced technical skills and equipment. In these attacks, chemical processes or precision equipment can be used to physically remove

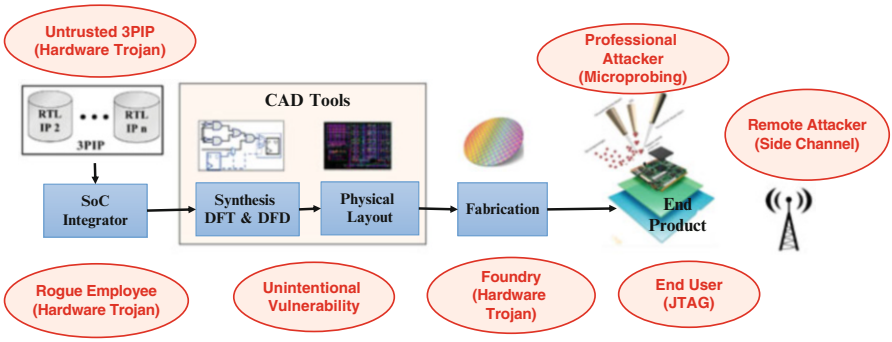


Fig. 2.2 Potential adversaries in different stages of SoC design process

micrometer thin layers of the device. Micro-probes can then be used to read values on data buses or inject faults into internal nets in the device to activate specific parts and extract information. Such attacks are usually invasive, require total access to the device, and result in destruction of the device.

2.2.3 Potential Adversary for Intentional Attacks

It is important to understand the potential adversaries who would utilize the security vulnerabilities to perform attacks. This may help designers in comprehending adversaries' capabilities and choosing right countermeasures depending on the target adversary. Adversary might be an individual or a party who intend to acquire, damage, or disrupt an asset for which he/she does not have permission to access. Considering an integrated circuit design process and entities involved in it, adversaries can be categorized into *insiders* and *outsiders*. Figure 2.2 shows the potential adversaries in different stages of SoC design process.

Insiders The design and manufacturing of integrated circuits have become more sophisticated and globally distributed with a higher possibility of being attacked by insiders who understand details of the design. An insider could be a rogue employee who work for design house, system integrator or could be a untrusted 3PIP, or a foundry. An insider:

- Has direct access to the SoC design either as an RTL or gate-level netlist, or as a GDSII layout file.
- Has high technical knowledge from the assumption that he/she is employed by a company in the IC design and supply chain.
- Has the capability to make modifications to the design, e.g., inserting hardware Trojans [9, 18]. These hardware Trojans can cause denial of service or create backdoors in the design through which sensitive information can be leaked. Other possible insider attacks are reduction in circuit reliability by manipulating circuit parameters, asset leakage, etc.

Outsiders This class of attackers are assumed to have access to end products in the market (e.g., a packaged IC). Outsider attackers can be divided into three groups based on their capabilities:

- **Remote hacker:** These attackers have no physical access to the device and must employ remote attacks described in Sect. 2.2.2, although they may have physical access to a similar device to develop their attack strategy. These attackers generally rely on exploiting software/hardware vulnerabilities, user errors, and design bugs to gain access to assets.
- **End user:** This group of attackers typically aim to gain free access to contents and services. In this case, the curious end users may rely on techniques already developed by professional attackers to carry their attack. For example, some hobbyists may find a way to jailbreak iPhone or Xbox gaming consoles and post the procedure on social media allowing end users with less expertise to duplicate the process. Jailbreaking allows users to install jailbreak programs and make Apple or Microsoft lose profits [19].
- **Professional attacker:** The most technically capable attackers are security experts or state-sponsored attackers whose motives are driven by financial or political reasons. These groups are capable of executing all types of attacks, including the more expensive invasive attacks described in Sect. 2.2.2, which requires removing the chip package and probing internal signals.

An *Insider* can more easily introduce or exploit the vulnerabilities in a design compared to an *outsider*. The main challenge for an *outsider* to perform an attack is that the internal functionality of the design is not known to the attacker. An *outsider* can reverse engineer the functionality of a chip but this technique would require extensive resource and time.

2.3 DSeRC: Design Security Rule Check

In order to identify and evaluate vulnerabilities associated with ICs, Design Security Rule Check (DSeRC) framework is to be integrated into the conventional digital IC design flow, as shown in Fig. 2.3. DSeRC framework will read the design files, constraints, and user input data, and check for vulnerabilities at all levels of abstraction (RTL, gate level, and physical layout level). Each of the vulnerabilities is to be tied with a set of rules and metrics so that each design's security can be quantitatively measured. At RTL, the DSeRC framework will assess the security of IPs which are either developed in-house or procured from third party (3P); and will provide feedback to design engineers so that the identified security issues can be addressed. After resolving the security vulnerabilities at RTL, the design will be synthesized to gate level and design-for-test (DFT) and design-for-debug (DFD) structures will be inserted. Then, DSeRC framework will analyze the gate-level netlist for security vulnerabilities. The same process will continue for the physical layout design as well. Through this process, the DSeRC framework will

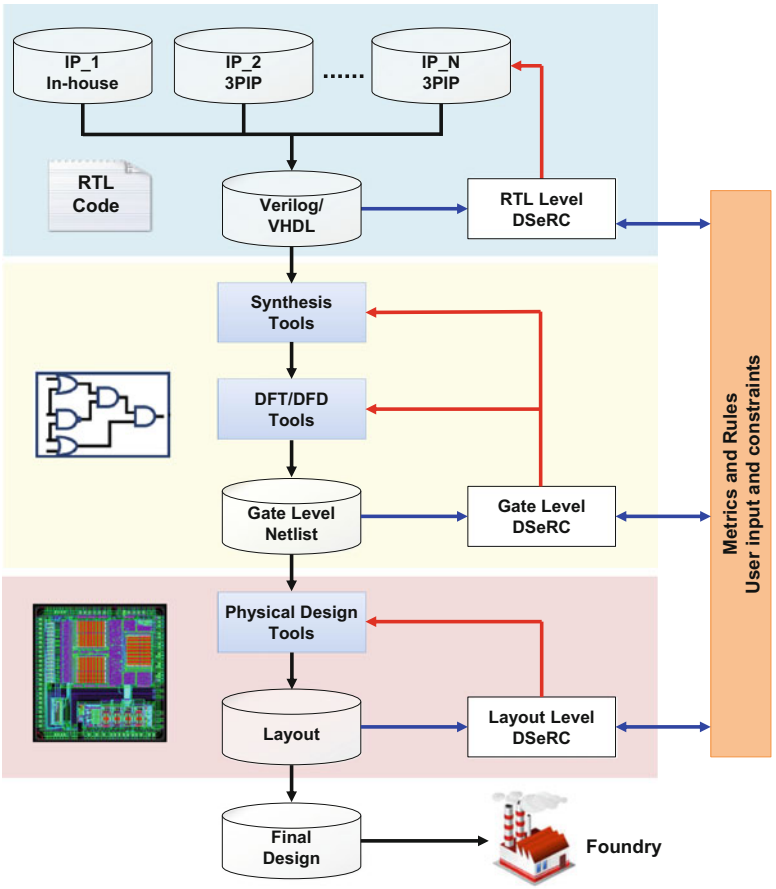


Fig. 2.3 DSeRC framework

allow the designers to identify and address security vulnerabilities at earliest design steps. This will significantly improve the security of ICs as well as considerably reduce the development time and cost by reducing the time-to-market constraint. Also, the DSeRC framework will allow the designer to quantitatively compare different implementation of same design and thereby allow the designer to optimize performance without compromising security.

The DSeRC framework will need some input from the designer. For example, the security assets need to be specified by hardware designers based on the target specifications of a design. The security vulnerabilities and the corresponding metrics and rules required for the development of the DSeRC framework are discussed in the following subsections.

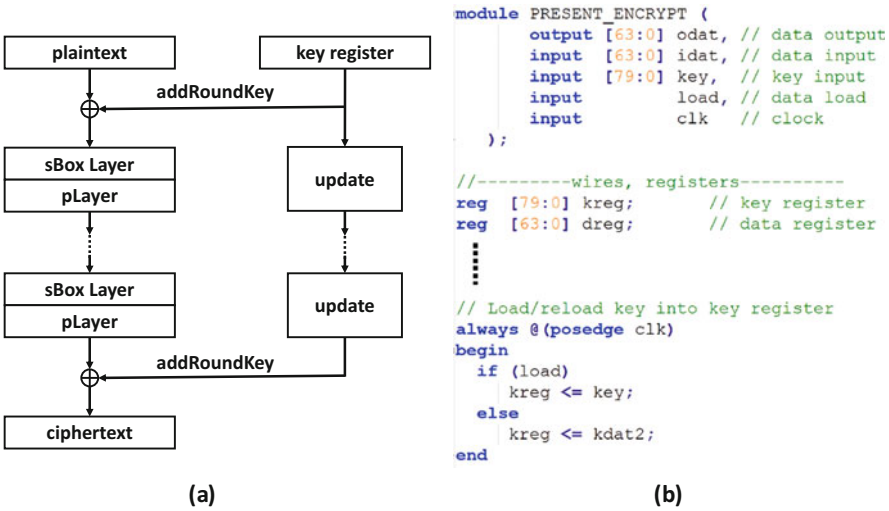


Fig. 2.4 Unintentional vulnerabilities created by design mistakes. (**a**) top-level description of PRESENT [20], (**b**) verilog implementation of PRESENT (http://opencores.org/)

2.3.1 Vulnerabilities

2.3.1.1 Sources of Vulnerabilities

Vulnerability in an IC means a weakness which allows an adversary to access the assets by performing some form of attack. Sources of the vulnerabilities in ICs are divided into following categories:

Vulnerabilities Due to Design Mistakes Traditionally the design objectives are driven by cost, performance, and time-to-market constraints; while security is generally neglected during the design phase. Additionally, security-aware design practices do not yet exist. Thus, IPs developed by different designers and design teams may present different set of vulnerabilities. We illustrate this further with a case study below.

Figure 2.4a shows the top-level description of PRESENT encryption algorithm [20]. A segment of its Verilog implementation is shown in Fig. 2.4b. We can see that the key is directly being assigned to the register, defined as "kreg" in the module. Although the encryption algorithm itself is secure, a vulnerability is unintentionally created in its hardware implementation. When this design is implemented, the "kreg" register will be included in the scan chain and an attacker can gain access to key through scan chain based attack [4].

Also, different implementation style of a same algorithm can have different levels of security. In a recent study [21], two AES SBox architectures, PPRM1 [22] and Boyar-Peralta [23], have been analyzed to evaluate which design is more susceptible to fault injection attack. The analysis showed that P-AES is more vulnerable to fault injection attack than the B-AES architecture.

Vulnerabilities Introduced by CAD Tools In the IC design process, CAD tools are extensively used for synthesis, DFT insertion, automatic place and route, etc. These tools are not equipped with understanding of the security vulnerabilities in ICs and can therefore introduce additional vulnerabilities in the design. As an example, synthesis tools can create new vulnerabilities in a design when the tool synthesizes the design from RTL to gate level. In the RTL specification of a finite state machine (FSM) there are don't-care conditions where the next state or the output of a transition is not specified. During the synthesis process, the synthesis tool tries to optimize the design by introducing deterministic states and transitions for the don't-care conditions. The introduction of don't-care states and transitions by the CAD tool can create vulnerability in the circuit by allowing a protected state to be illegally accessed through the don't-care states and transitions [8].

We use the controller circuit of an AES encryption module (http://opencores.org/) as another case study to demonstrate the vulnerability introduced by the CAD tools. The state transition diagram of the FSM shown in Fig. 2.5b implements the AES encryption algorithm on the data path shown in Fig. 2.5a. The FSM is composed of 5 states and each of these states controls specific modules during the ten rounds of AES encryption. After ten rounds, the "Final Round" state is reached and the FSM generates the control signal *finished* = 1 which stores the result of the "Add Key" module (i.e., the ciphertext) in the "Result Register." For this FSM, the "Final Round" is a protected state because if an attacker can gain access to the "Final Round" without going through the "Do Round" state, then premature results will be stored in "Result Register," potentially leaking the secret key. Now, during the synthesis process if a don't-care state is introduced that has direct access to a protected state, then it can create vulnerability in the FSM by allowing the attacker to utilize this don't-care state to access the protected state. Let us consider that the "Don't-Care_1" state shown in Fig. 2.5b is introduced by the synthesis tool and this state has direct access to the protected state "Final Round." Introduction of "Don't-Care_1" state represents a vulnerability introduced by the CAD tool because this don't-care state can facilitate fault and Trojan based attack. For example, an attacker can inject a fault to go to state "Don't Care_1" and access the protected state "Final Round" from this state. The attacker can also utilize the "Don't Care_1" to implant a Trojan. The presence of this don't-care state gives the attacker a unique advantage because this state is not taken into consideration during validation and testing; therefore it is easier for the Trojan to evade detection.

Additionally, during the synthesis process CAD tools flatten all the modules of the design together and try to optimize the design for power, timing, and/or area. If a secure module (e.g., encryption module) is present in an SoC, design flattening and the multiple optimization processes can lead to merging trusted blocks with those untrusted. These design steps, which the designer has little control of, can introduce vulnerabilities and cause information leakage [24].

Vulnerabilities Introduced by DFT and DFD Structure High testability is important for critical systems to ensure proper functionality and reliability throughout their lifetime. Testability is a measure of controllability and observability of

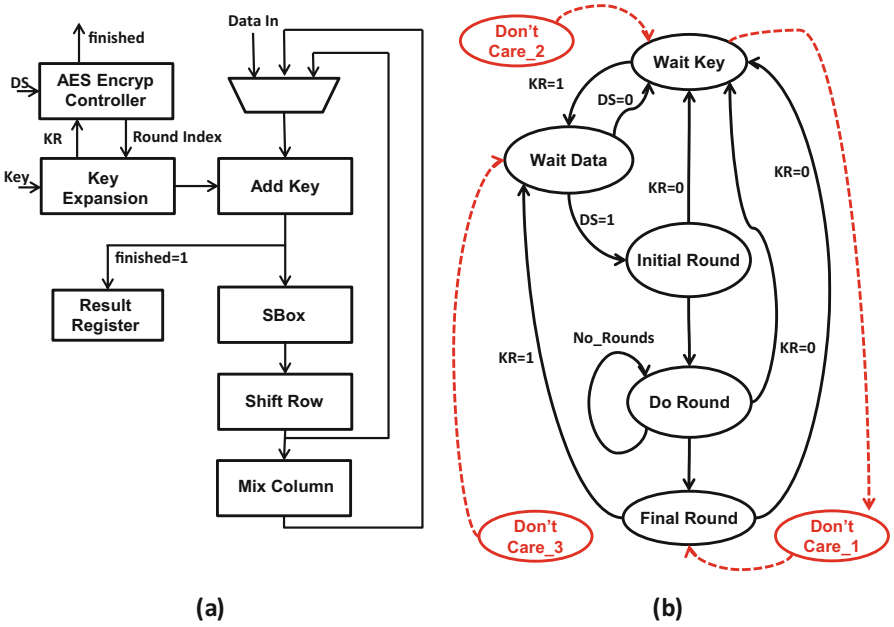


Fig. 2.5 Unintentional vulnerabilities created by CAD tools. (**a**) and (**b**) show data path and finite state machine (FSM) of AES encryption module. KR and DS stand for Key Ready and Data Stable signal, respectively; the red marked states and transitions represent the don't-care states and transitions introduced by the CAD tool

signals (i.e., nets) in a circuit. Controllability is defined as the difficulty of setting a particular logic signal to "1" or "0" and observability is defined as the difficulty of observing the state of a logic signal. To increase testability and debug, it is very common to integrate design-for-test (DFT) and design-for-debug (DFD) structures in a complex design. However, the increased controllability and observability added by DFT and DFD structures can create numerous vulnerabilities by allowing attackers to control or observe internal states of an IC [25].

In general, test and debug can be viewed as the opposite of security when it comes to accessing circuit internals, as shown in Fig. 2.6. Unfortunately, the DFT and DFD structures cannot be simply avoided in modern designs because of large amount of unexpected defects and errors that occur during the fabrication. Additionally, National Institute of Standards and Technology (NIST) requires that any design used in critical applications needs to be properly testable, both in pre- and post-manufacturing. Therefore, the DFT and DFD structures must be incorporated in ICs, though these structures may create vulnerability. Thus, it is necessary for the DSeRC framework to check whether any security vulnerability is introduced by the DFT and DFD.

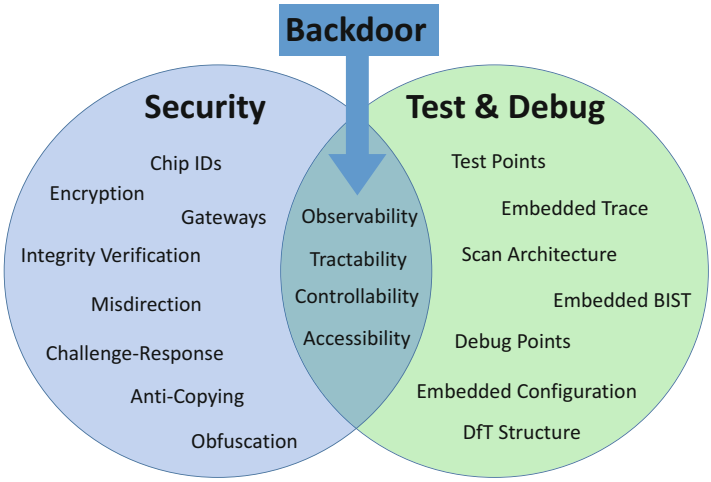


Fig. 2.6 Requirement for high quality test and debug contradicts security

2.3.1.2 Vulnerabilities at Different Abstraction Levels

For the development of DSeRC framework, each vulnerability needs to be assigned to one or multiple proper abstraction levels where it can be identified efficiently. Generally, an IC design flow will go through specification, RTL design, gate-level design and consequently physical layout design. DSeRC framework aims at identifying vulnerabilities as early as possible during the design flow because late evaluation can lead to a long development cycle and high design cost. Also, vulnerabilities in one stage, if not addressed, may introduce additional vulnerabilities during transitions from one level to the next. In this section, we categorize vulnerabilities based on the abstraction levels (see Table 2.1).

Register Transfer Level (RTL) The design specification is first described in a hardware description language (HDL) (e.g., verilog) to create the RTL abstraction of the design. Several attacks performed at the RTL have been proposed in literature. For example, Fern et al. [26] demonstrated that don't-care assignments in RTL code can be leveraged to implement hardware Trojans that leak assets. In this case, the don't-care assignments in RTL code is the source of vulnerability. Additionally, in the RTL, hardware Trojans are most likely to be inserted at hard-to-control and hard-to-observe parts of the code [27]. Identifying hard-to-control and hard-to-observe parts of the code can help designers assess the susceptibility of the design to Trojan insertion at the RTL.

In general, vulnerabilities identified at the RTL are comparatively easier to address. However, some vulnerabilities, e.g., how susceptible the design is to fault or side-channel attacks, are very challenging if not impossible to identify at this level.

Gate Level The RTL specification is synthesized into gate-level netlist using a synthesis tools, e.g., design compiler. At gate level, a design usually is represented with a flattened netlist hence, loses its abstraction; however, more accurate information about the implementation in term of gates or transistors are available. At gate level, hard-to-control and hard-to-observe nets can be used to design hard-to-detect Hardware Trojans [28]. Also, transition from RTL to gate level can introduce additional vulnerabilities by the CAD tools. Examples of vulnerabilities introduced by the tools have been discussed in Sect. 2.3.1. These vulnerabilities need to be analyzed at the gate level.

DFT and DFD structures are generally incorporated in the ICs at the gate level. Therefore, the vulnerabilities introduced by the test and debug structure (see Sect. 2.3.1) need to be analyzed at the gate level.

Layout Level Physical layout design is the last design stage before shipping the chip to fabrication, so all the remaining vulnerabilities should be addressed in this level. During layout design, the placement and routing phase gives information about the spatial arrangements of the cells and metal connections in the circuit. In the layout level, power consumption, electromagnetic emanations, and execution time can be accurately modeled. Therefore, vulnerability analysis of side-channel and fault-injection based attacks can be done very accurately at this level. Additionally, some vulnerability analyses, e.g., vulnerability to probing attack [29] can only be done at the layout level. However, any analysis done in this level is very time consuming compared to RTL and gate level.

2.3.2 Metrics and Rules

The vulnerabilities that have been discussed so far are to be tied with metrics and rules so that each design's security can be quantitatively measured (see Table 2.1). These rules and metrics of the DSeRC framework can be compared with the design rule check (DRC). In DRC, semiconductor manufacturers convert manufacturing specifications into a series of metrics that enable the designer to quantitatively measure a mask's manufacturability. For the DSeRC framework, each vulnerability needs to be mathematically modeled and the corresponding rule and metric need to be developed so that the vulnerability of a design can be quantitatively evaluated. As for the rules, there can be two types of rules; one type is based on *quantitative metric* and the other type is based on *a binary classification* (yes/no).

We present a brief description of some of the rules and metrics corresponding to the vulnerabilities shown in Table 2.1.

Asset Leakage As discussed in Sect. 2.3.1.1, vulnerabilities associated asset leakage can be unintentionally created by design-for-test (DFT), design-for-debug (DFD) structures, CAD tools, and/or by designer's mistake. These vulnerabilities cause violation of information security policies, i.e., confidentiality and integrity policies. Therefore, the metric for identifying these vulnerabilities is *confidentiality*

Table 2.1 Vulnerabilities, metrics, and rules included in DSeRC

	Vulnerability	Metric	Rule	Attack (Attacker)
RTL Level	Dangerous Don't Cares	Identify all "X" assignments and check if "X" can propagate to observable nodes	"X" assignments should not be propagated to observable nodes	Hardware Trojan Insertion (Insider)
	Hard-to-control & hard-to-observe signal	Statement hardness and signal observability [27]	Statement hardness (signal observability) should be lower (higher) than a threshold value	Hardware Trojan (Insider)
	Asset leakage	Structure checking and information flow tracking	YES/NO: access assets or observe assets	Asset hacking (End user)
	...	...	...	...
Gate Level	Hard-to-control & hard-to-observe net	Net controllability and observability [28]	Controllability and observability should be higher than a threshold value	Hardware Trojan (Insider)
	Vulnerable Finite State Machine (FSM)	Vulnerability factor of fault injection (VF_{FI}) & vulnerability factor of Trojan insertion (VF_{Tro}) [8]	VF_{FI} and VF_{Tro} should be zero	Fault injection, Hardware Trojan (Insider, end user)
	Asset leakage	Confidentiality and integrity assessment [30]	YES/NO: access assets or observe assets	Asset hacking (End user)
	Design-for-Test (DFT)	Confidentiality and integrity assessment [30]	YES/NO: access assets or observe assets	Asset hacking (End user)
	Design-for-Debug (DFD)	Confidentiality and integrity assessment [30]	YES/NO: access assets or observe assets	Asset hacking (End user)
	...	...	...	...
Layout Level	Side-channel signal	Side-channel vulnerability factor (SVF) [31]	The SVF should be lower than a threshold value	Side-channel attack (End user)
	Micro-probing	Exposed area of the security-critical nets which are vulnerable to micro-probing attack [29]	The exposed area should be lower than a threshold value	Micro-probing attack (Professional attacker)
	Injectable Fault/Error	Timing violation vulnerability factor (TVVF) [21]	TVVF higher than a threshold means the implementation is insecure	Timing based Fault injection attack (End user)
	...	...	...	...

and integrity assessment. In [30], authors have presented a framework that validates if confidentiality and integrity policies are being maintained in the SoC. This framework is based on a novel concept of modeling an asset (e.g., a net carrying a secret key) as a fault and leveraging the known test algorithms to detect that fault. A successful detection of the fault indicates that there is flow of information from the asset to an observable point. The rule for this vulnerability will be whether the asset value can be propagated to any observable points. If the assessment result is YES, then there exits vulnerability in the design and the design is not secure.

Vulnerable FSM The synthesis process of a finite state machine (FSM) can introduce additional security risks in the implemented circuit by inserting additional don't-care states and transitions. An attacker can utilize these don't-care states and transitions to facilitate fault injection and Trojan attacks. In [8], authors have presented two metrics, named vulnerability factor of fault injection (VF_{FI}) and vulnerability factor of Trojan insertion (VF_{Tro}) to quantitatively analyze how susceptible an FSM is against fault injection and Trojan attacks, respectively. The higher the values of these two metrics are, the more vulnerable the FSM is to fault and Trojan attacks. For this vulnerability, the rule can be stated as follows; for an FSM design to be secured against fault injection and Trojan insertion attack, the values of VF_{FI} and VF_{Tro} should be zero.

Micro-Probing Attack Micro-probing is one kind of physical attack that directly probes the signal wires in order to extract sensitive information. This attack has raised serious concerns for security critical applications. In [29], authors have presented a layout-driven framework to quantitatively evaluate a post place-and-route design in terms of exposed area of the security-critical nets which are vulnerable to micro-probing attack. The larger the exposed area, the more vulnerable the SoC is to probing attack. Therefore, the rule for micro-probing vulnerability can be stated as follows, the exposed area to micro-probing should be lower than a threshold value.

Susceptibility to Trojan Insertion In [27], authors have presented a metric named "Statement Hardness" to evaluate the difficulty of executing a statement in the RTL code. Areas in a circuit with large value of "Statement Hardness" are more vulnerable to Trojan insertion. Therefore, the metric "Statement Hardness" gives the quantitative measure of a design's susceptibility to Trojan insertion. Next is to define rule(s) to evaluate if the design is secure. For this vulnerability, the rule can be stated as follows; for a design to be secured against Trojan insertion attack, statement hardness of each statement in the design should be lower than SH_{thr}. Here, SH_{thr} is a threshold value that needs to be derived from the area and performance budget.

At gate level, a design is vulnerable to Trojan insertion which can be implemented by adding and deleting gates. To hide the effect of inserted Trojan, an adversary will target *hard-to-detect* areas of the gate-level netlist. *Hard-to-detect* nets are defined as, nets which have low transition probability and are not testable through well-known fault testing techniques (stuck-at, transition delay, path delay,

and bridging faults) [28]. Inserting a Trojan in *hard-to-detect* areas would reduce the probability to trigger the Trojan and thereby, reduce the probability of being detected during verification and validation testing. In [32], authors have proposed metrics to evaluate *hard-to-detect* areas in the gate-level netlist.

Fault and Side-Channel Attacks Authors in [21] have introduced Timing Violation Vulnerability Factor (TVVF) to evaluate the vulnerability of a hardware structure to setup time violation attacks which are one subset of fault-injection attacks. In [33], authors have proposed a framework named AMASIVE (Adaptable Modular Autonomous Side-Channel Vulnerability Evaluator) to automatically identify side-channel vulnerabilities of a design. Also, a metric named side-channel vulnerability factor (SVF) has been proposed to evaluate the ICs' vulnerability to power side-channel attacks [31].

Note that the development of the metrics is a challenging task because ideally the metrics must be independent of attack model and application/functionality of a design. For example, an attacker can apply voltage starving or clock glitching based fault injection attacks to obtain the private key of AES or RSA encryption modules. The metric for fault injection needs to provide a quantitative measure of vulnerability for any design (AES or RSA) against any of these attacks (voltage starving or clock glitching). One strategy would be to first identify the root vulnerability that these attacks try to exploit. For this particular example, both voltage starving and clock glitching try to exploit the setup time violation, a common criteria for success of both attacks. Then the framework must evaluate the difficulty of violating setup time for a given design to gain access to the target security assets.

2.3.3 Workflow of DSeRC Framework

In this section we describe how the rules and metrics will be used to identify a vulnerability under DSeRC framework. Table 2.1 shows the vulnerabilities and their corresponding metrics and rules that our DSeRC framework covers. The example of PRESENT encryption algorithm will be used to illustrate the workflow of DSeRC framework. The designer will first give the RTL design files and the name of the asset (i.e., "key") as input to DSeRC framework. DSeRC framework will use the information flow tracking to analyze if the "key" can be leaked through any observation points (e.g., registers). As for this design the "key" can be observed through "kreg" register. Therefore, the framework will give a preemptive warning to the designer that if the register "kreg" is included in the DFT structure then, the key can be leaked through scan chain. It will be up to the designer to apply a countermeasure to address this vulnerability before moving to next level of abstraction. One possible countermeasure would be is to exclude "kreg" from the scan chain.

After addressing this vulnerability, the design can be synthesized and DFT inserted. The designer will then give the synthesized gate-level netlist to DSeRC and the framework will use the *confidentiality assessment* [30] technique of DSeRC framework to analyze if the "key" can be leaked through any observable point. If the "key" cannot be leaked, then the DSeRC rule will be satisfied and the design can be considered to be secured against asset leakage. On the other hand, if the DSeRC framework identifies the "key" is being leaked to scan flip-flops (observable points), then the framework will raise flag and point the scan flip-flops which carry information about the key. The designer needs to address this vulnerability before moving to physical layout. One possible approach would to apply secure scan structure [34, 35] to counter the vulnerability introduced by the DFT structure.

Note that manually tracking the asset in an SOC to evaluate if it is being leaked through an observable point would be an exhaustive if not impossible task for the designer. On the other hand, DSeRC framework will be able to pinpoint the asset leakage paths allowing the designers to concentrate their analysis effort on those paths and make informed decision.

Some vulnerability can have the common factors, which provide an opportunity to merge multiple vulnerability analyses. As shown in Table 2.1, both asset leakage and fault injection need a metric to evaluate the observability of a net in the circuit structure. Therefore, observability analysis can be executed once for both vulnerability analyses.

While DSeRC may be a logical and necessary step in designing secure ICs, it would not eliminate the need for security subject matter expert. The DSeRC framework is intended to be an automated framework and therefore, may not take the application and use cases of an IC into consideration. For example, the Trojan observability metric will report the difficulty of observing each signal in the design. For Trojan detection, a high observability metric is desired. However, for an encryption module, a high observability metric for the private key will be a serious threat. Therefore, it would be upto the designer to interpret the results generated by the DSeRC framework.

2.4 Development of DSeRC Framework

2.4.1 Vulnerabilities, Metrics, and Rules

Vulnerability identification is crucial for the development of DSeRC framework. As more attacks are developed, the vulnerabilities exploited by these new attacks need to be identified. And as more vulnerabilities are discovered, their corresponding metrics and rules need to be developed as well. Given the diversity and vastness of security threats, it will take the combined effort of academic and industry researchers to develop a comprehensive set of vulnerabilities and corresponding rules and metrics for the DSeRC framework.

2.4.2 Tool Development

The DSeRC framework is intended to be integrated with the traditional IC design flow so that security evaluation can be made as an inherent part of the design process. This requires the development of CAD tools which can automatically evaluate the security of a design based on DSeRC rules and metrics. The tools' evaluation times need to be scalable with the design size. Also, the tools should be easy to use and the outputs generated by the tools need to be understandable by the design engineer.

2.4.3 Development of Design Guidelines for Security

To avoid some common security problems in early design stages, good design practices learned through experience can be used as guidelines. The design guideline is able to guide design engineers what to do (Do-s) and not to do (Don't-s) during the initial design. Do-s and Don't-s are very common in VLSI design and test domain, in order to improve the design quality and testability of faults [36]. These Do-s and Don't-s, such as initializable flip-flops, no asynchronous logic feedback, and no gating of clocks to scan flip-flops, are very straight-forward, but quite effective to make a design testable and thus save time and cost in design cycles. This is also applicable to hardware security. Taking the PRESENT in Fig. 2.4 as an example, if "kreg" does not become part of scan chain, the key leakage problem can be addressed at the design stage. However, to date, no comprehensive design guideline has been developed for hardware security. This can be a research direction where academia and industry can explore.

2.4.4 Development of Countermeasure Techniques

Knowledge of the vulnerabilities of a design is not sufficient to protect it. A series of low-cost countermeasure techniques are also needed for each vulnerability. One additional extension of the DSeRC framework will be to provide low-cost mitigation techniques for each vulnerability. The suggested countermeasure techniques will be a good hint to design engineers who lack of knowledge in hardware security. The improvement in security after applying the countermeasures can be measured by running the DSeRC check again.

As an example, in [8] authors have proposed a countermeasure to address the vulnerabilities introduced in FSMs by the synthesis tool or design mistake. In their proposed approach the state FFs are to be replaced by "Programmable State FFs." "Programmable State FFs" are defined as the state FFs which go to the Reset/Initial state if the protected state is tried to be accessed by any other state apart from the authorized states. The detailed architecture of the "Programmable State FFs" has been discussed in [8].

2.5 Conclusion

In this chapter, we have presented the basic concept of Design Security Rule Check (DSeRC) to analyze vulnerabilities in a design and consequently assess its security level at design stage. One of the main challenges for the development of the DSeRC framework is associated with mathematically modeling vulnerabilities for quantitative evaluation. This is because DSeRC framework needs to validate the security of a design regardless of the application of the design or attack models. Although DSeRC will not eliminate the need for security subject matter experts, it will, however, expedite the security analysis of ICs and SoCs, and increase the design engineer's awareness to the security issues of their designs.

References

1. P.C. Kocher, Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems, in *Annual International Cryptology Conference* (Springer, Berlin/Heidelberg, 1996), pp. 104–113
2. P.C. Kocher, J. Jaffe, B. Jun, Differential power analysis, in *CRYPTO* (1999)
3. D. Hely et al., Scan design and secure chip [secure IC testing], in *Proceedings of the 10th IEEE IOLTS* (July 2004), pp. 219–224
4. J. Lee, M. Tehranipoor, C. Patel, J. Plusquellic, Securing scan design using lock and key technique, in *Proceedings - IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)* (2005)
5. B. Yang, K. Wu, R. Karri, Scan-based side-channel attack on dedicated hardware implementations of data encryption standard, in *Proceedings of International Test Conference* (2004)
6. E. Biham, A. Shamir, Differential fault analysis of secret key cryptosystems, in *CRYPTO* (1997)
7. C. Dunbar, G. Qu, Designing trusted embedded systems from finite state machines. ACM Trans. Embed. Comput. Syst. **13**(5s), 1–20 (2014)
8. A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, M. Tehranipoor, AVFSM: a framework for identifying and mitigating vulnerabilities in FSMs, in *Proceedings of the 53rd Annual Design Automation Conference* (ACM, 2016), p. 89
9. M. Tehranipoor, F. Koushanfar, A survey of hardware Trojan taxonomy and detection. IEEE Des. Test Comput. **27**, 10–25 (2010)
10. ARM inc., Building a secure system using TrustZone technology, http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf
11. K. Xiao, A. Nahiyan, M. Tehranipoor, Security rule checking in IC design. Computer **49**(8), 54–61 (2016)
12. Eric Peeters, SoC security architecture: current practices and emerging needs, in *Design Automation Conference (DAC)* (IEEE, 2015), pp. 1–6
13. P. Kocher, Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems, in *Advances in Cryptology* (1996), pp. 104–113
14. T. Korak, T. Plos, Applying remote side-channel analysis attacks on a security-enabled NFC tag, in *Topics in cryptology–CT-RSA 2013* (February 2013)
15. A. Das, J. Da Rolt, S. Ghosh, S. Seys, S. Dupuis, G. Di Natale et al., Secure JTAG implementation using Schnorr protocol. J. Electron. Test. Theory Appl. **29**(2), 193–209 (2013)

16. S. Chen et al., Security vulnerabilities: from analysis to detection and masking techniques. Proc. IEEE **94**(2), 407–418 (2006)
17. S.P. Skorobogatov, Semi-invasive attacks - a new approach to hardware security analysis, in Technical Report UCAM-CL-TR-630. University of Cambridge Computer Laboratory, April 2005
18. M. Tehranipoor, C. Wang, *Introduction to Hardware Security and Trust* (Springer, New York, 2011)
19. M.A. Harris, K.P. Patten, Mobile device security considerations for small-and medium-sized enterprise business mobility, Inf. Manage. Comput. Secur. **22**, 97–114 (2014)
20. A. Bogdanov et al., PRESENT: an ultra-lightweight block cipher, in *Cryptographic Hardware and Embedded Systems* (2007)
21. B. Yuce, N. Ghalaty, P. Schaumont, TVVF: estimating the vulnerability of hardware cryptosystems against timing violation attacks, in *Hardware Oriented Security and Trust* (2015)
22. S. Morioka, A. Satoh, An optimized s-box circuit architecture for low power AES design, in *Cryptographic Hardware and Embedded Systems* (2003)
23. J. Boyar, R. Peralta, A small depth-16 circuit for the AES s-box, in *Proceedings of Information Security and Privacy Research* (2012)
24. T. Huffmire et al., Moats and drawbridges: an isolation primitive for reconfigurable hardware based systems, in *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007)
25. J. Rolt et al., Test versus security: past and present. IEEE Trans. Emerg. Top. Comput. **2**(1), 50–62 (2013)
26. N. Fern, S. Kulkarni, K. Cheng, Hardware Trojans hidden in RTL don't cares - automated insertion and prevention methodologies, in *International Test Conference (ITC)* (2015)
27. H. Salmani, M. Tehranipoor, Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level, in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (2013), pp. 190–195
28. H. Salmani, R. Karri, M. Tehranipoor, On design vulnerability analysis and trust benchmarks development, in *Proceedings of IEEE 31st International Conference on Computer Design (ICCD)* (2013), pp. 471–474
29. Q. Shi, N. Asadizanjani, D. Forte, M. Tehranipoor, A layout-driven framework to assess vulnerability of ICs to microprobing attacks, in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (IEEE, 2016), pp. 155–160
30. G. Contreras, A. Nahiyan, S. Bhunia, D. Forte, M. Tehranipoor, Security vulnerability analysis of design-for-test exploits for asset protection, in *Asia and South Pacific Design Automation Conference* (2017, to appear)
31. J. Demme et al., Side-channel vulnerability factor: a metric for measuring information leakage, in *39th Annual International Symposium on Computer Architecture* (2012), pp. 106–117
32. M. Tehranipoor, H. Salmani, X. Zhang, *Integrated Circuit Authentication: Hardware Trojans and Counterfeit Detection* (Springer, Cham, 2013)
33. S.A. Huss, M. Stottinger, M. Zohner, AMASIVE: an adaptable and modular autonomous side-channel vulnerability evaluation framework, in *Number Theory and Cryptography* (Springer, Berlin, Heidelberg, 2013), pp. 151–165
34. J. Lee, M. Tehranipoor, C. Patel, J. Plusquellic, Securing designs against scan-based side-channel attacks. IEEE Trans. Dependable Secure Comput. **4**(4), 325–336 (2007)
35. J. Lee, M. Tehranipoor, J. Plusquellic, A low-cost solution for protecting IPs against scan-based side-channel attacks, in *VLSI Test Symposium* (2006)
36. M. Bushnell, V. Agrawal, in *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits* (Springer, New York, 2000)

Chapter 3
Digital Circuit Vulnerabilities to Hardware Trojans

Hassan Salmani and Mark Tehranipoor

3.1 Introduction

Adopted in the interest of economy, the horizontal integrated circuit design process has raised serious concerns for national security and critical infrastructures [1, 2]. An adversary is afforded plenty of opportunities to interfere with its design process, and circuit parametric or functional specifications may be jeopardized, allowing malicious activities [3–6]. Any intentional modification of design specifications and functionality to undermine its characteristics and correctness is called a hardware Trojan.

Hardware Trojans can be realized by including additional circuits at the register transfer or gate-level or by changing circuit parameters like wire thickness or component size at the layout-level, to name a few. Hardware Trojans can reduce circuit reliability, change or disable its functionality at a certain time, reveal its detailed implementation, or grant covert access to unauthorized entities [7, 8]. For instance, a third party intellectual property (IP) provider can enclose extra circuitry within a cryptographic module at the gate-level to leak its secret key.

A number of proposed approaches facilitate hardware Trojan detection by analyzing circuit side-channel signals or by increasing the probability of Trojan full activation. Incurred extra switching activity or induced additional wiring and gate capacitance affects circuit side-channel signals such as power and delay. Path delay fingerprint and delay measurement based on shadow registers are techniques intended to capture Trojan impact on circuit delay characteristics [9, 10]. Transient

H. Salmani (✉)
Howard University, Washington, DC, USA
e-mail: hassan.salmani@howard.edu

M. Tehranipoor
University of Florida, Gainesville, FL, USA
e-mail: tehranipoor@ece.ufl.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_3

current integration, circuit power fingerprint, and static current analysis are power-based Trojan detection techniques [11–13]. Efficient pattern generation is also necessary to discover a Trojan's impact upon circuit characteristics beyond process and environmental variations [14, 15]. In addition to new pattern generation techniques, design-for-hardware-trust methodologies have been proposed to increase switching activity inside a Trojan circuit while reducing main circuit switching activity acting as background noise, to enhance Trojan detection resolution [3, 4, 6].

At the gate-level, hardware Trojan triggers might be driven by signals with low transition probabilities to reduce Trojan contribution into circuit specifications. As a circuit layout is available to an untrusted foundry for fabrication, a hardware Trojan might be inserted in existing white spaces to leave circuit layout size unchanged. Therefore, a systematic approach is required to analyze the susceptibility of gate-level netlists and circuit layouts to hardware Trojan insertion to identify and quantify the vulnerability of a signal or a section to hardware Trojan insertion and attacks.

3.2 The Gate-Level Design Vulnerability Analysis Flow

Functional hardware Trojans are realized by adding or removing gates; therefore, the inclusion of Trojan gates or the elimination of circuit gates affects circuit side-channel signals such as power consumption and delay characteristics, as well as the functionality. To minimize a Trojan's contribution to the circuit side-channel signals, an adversary can exploit hard-to-detect areas (e.g., nets) to implement the Trojan. Hard-to-detect areas are defined as areas in a circuit not testable by well-known fault-testing techniques (stuck-at, transition delay, path delay, and bridging faults) or not having noticeable impact on the circuit side-channel signals. Therefore, a vulnerability analysis flow is required to identify such hard-to-detect areas in a circuit. These areas provide opportunities to insert hard-to-detect Trojans and invite researchers to develop techniques to make it difficult for an adversary to insert Trojans.

Figure 3.1 shows the vulnerability analysis flow performing power, delay, and structural analyses on a circuit to extract the hard-to-detect areas. Any transition inside a Trojan circuit increases the overall transient power consumption; therefore, it is expected that Trojan inputs are supplied by nets with low transition probabilities to reduce activity inside the Trojan circuit.

The *Power Analysis* step in Fig. 3.1 is based on analyzing switching activity; it determines the transition probability of every net in the circuit assuming the probability of 0.5 for "0" or "1" at primary inputs and at memory cells' outputs. Then, nets with transition probabilities below a certain threshold are considered as possible Trojan inputs. The *Delay Analysis* step performs path delay measurement based on gates' capacitance. This allows to measure the additional delay induced by Trojan by knowing the added capacitance to circuit paths. The Delay Analysis step identifies nets on non-critical paths as they are more susceptible to Trojan insertion and harder to detect their changed delay. To further reduce Trojan impact on circuit

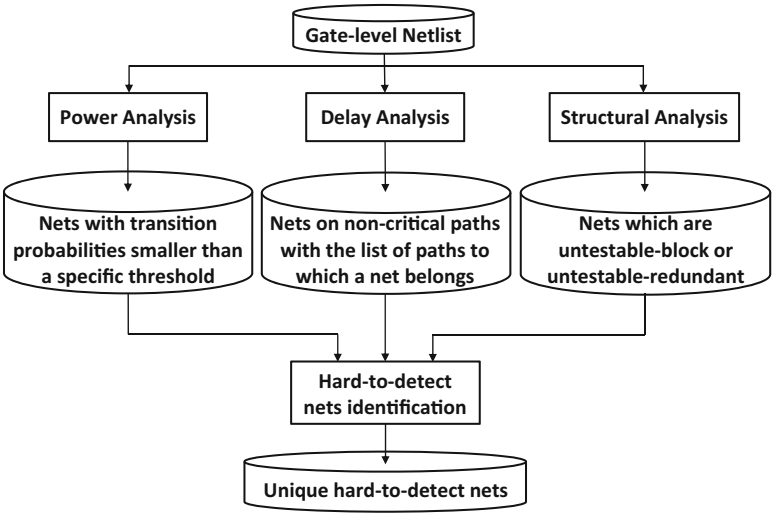


Fig. 3.1 The gate-level vulnerability analysis flow

delay characteristics, it also reports the paths to which a net belongs to avoid selecting nets belonging to different sections of one path. The *Structural Analysis* step executes the structural transition delay fault testing to find untestable blocked and untestable redundant nets. Untestable redundant nets are not testable because they are masked by a redundant logic, and they are not observable through primary output or scan cells. Untestable blocked nets are not controllable or observable by untestable redundant nets. Tapping Trojan inputs to untestable nets hides Trojan impact on delay variations.

At its end, the vulnerability analysis flow reports unique hard-to-detect nets that are the list of untestable nets with low transition probabilities and nets with low transition probabilities on non-critical paths while not sharing any common path. Note that when a Trojan impacts more than one path, it provides greater opportunities for detection. Avoiding shared paths makes a Trojan's contribution to affected paths' delay minimal, which can be masked by process variations, making it difficult to detect and distinguish the added delay from variations. The reported nets are ensured to be untestable by structural test patterns used in production tests. They also have low transition probabilities so Trojans will negligibly affect circuit power consumption. As the nets are chosen from non-critical paths without any shared segments, it would be extremely difficult to detect Trojans by delay-based techniques.

The vulnerability analysis flow can be implemented using most electronic design automation (EDA) tools, and the complexity of the analysis is linear with respect to the number of nets in the circuit. The flow is applied to the Ethernet MAC 10GE circuit [16], which implements 10Gbps Ethernet Media Access Control functions. Synthesized at 90nm Synopsys technology node, the Ethernet MAC 10GE circuit

consists of 102,047 components, including 21,830 flip-flops. The Power Analysis shows that out of 102,669 nets in the circuit, 23,783 of them have a transition probability smaller than 0.1, 7003 of them smaller than 0.01, 367 of them smaller than 0.001, and 99 of them smaller than 0.0001. The Delay Analysis indicates that the largest capacitance along a path, representing path delay, in the circuit is 0.065717825 pF, and there are 14,927 paths in the circuit whose path capacitance is smaller than 70 % of the largest capacitance, assuming that paths longer than 70 % in a circuit can be tested using testers. The Structural Analysis finds that there is no untestable fault in the circuit. By excluding nets sharing different segments of one path, there are 494 nets in the Ethernet MAC 10GE circuit considered to be areas where Trojan inputs could be used while ensuring the high difficulty of detection based on side-channel and functional test techniques.

3.3 The Layout-Level Design Vulnerability Analysis Flow

A physical design tool takes a synthesized netlist and associated technology library information and performs placement and routing considering design constraints such as performance, size, and manufacturability. Cells are typically placed in rows and their interconnections are realized through metal layers above the cells. Large circuits such as system-on-chips take larger areas for placement and require more metal layers for routing to meet circuit constraints, besides circuit functionality. However, a final circuit layout may contain a considerable amount of whitespaces in substrate and empty routing channels in metal layers above the substrate. These empty spaces can be used by an untrusted foundry to place and route Trojan cells with minimum impact on circuit specification. To study the vulnerability of a circuit layout to hardware Trojan insertion, Fig. 3.2 shows a novel circuit vulnerability analysis flow at the layout-level.

3.3.1 Cell and Routing Analyses

A gate-level synthesized netlist, along with design constraints and technology library information, is fed into a physical design tool for placement and routing. The circuit layout, the output of physical design, shows gates' location and their detail wiring through metal layers. In addition to the circuit layout, an updated design netlist with circuit parasitic parameters is obtained. The proposed flow includes two major steps: the *Cell Analysis* and the *Routing Analysis* to study cell distribution and routing congestion. The Cell Analysis step screens the circuit silicon to extract cells' location. It then determines whitespaces distribution and their size. The Routing Analysis step extracts used routing channels in each metal layer and determines unused ones.

www.allitebooks.com

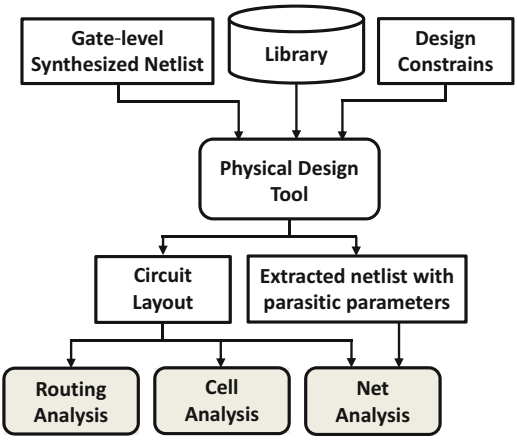


Fig. 3.2 The layout-level vulnerability analysis flow

After obtaining the circuit layout, the Cell Analysis obtains the circuit size and collects placed cells and their coordination by screening the circuit layout. Using these information, whitespaces in the circuit substrate are identified. Any whitespace whose area size is greater than that of the smallest cell in the technology library is considered a potential location for one or more Trojan cells insertion. The Cell Analysis also obtains the distribution of cells and whitespaces across the layout. The Routing Analysis collects used and unused routing channels in metal layers above the substrate. Available routing channels can potentially be used for Trojan cells interconnection and their connections to the main circuit. Similar to the Cell Analysis, the Routing Analysis also collects the distribution of used and empty routing channels in all metal layers. After determining whitespace and unused routing channels distributions of a circuit layout, the vulnerability of a region of circuit layout to hardware Trojan cells placement is defined as:

$$V(r) = WS(r) \times UR(r) \tag{3.1}$$

where $V(r)$ indicates the vulnerability of region r, $WS(r)$ the normalized whitespace of region r, and $UR(r)$ the normalized unused routing channels of region r. It is expected that Trojan cells are inserted in regions with high $V(r)$ where there are equally high $WS(r)$ and $UR(r)$.

While being inserted in a region with high V may not guarantee Trojan detection avoidance, to remain hidden from delay-based detection techniques, Trojans should be inserted in regions with enough whitespace and empty routing channels and tapped to nets on non-critical paths. The value of vulnerability to delay-resistant Trojans ($V_{\mathrm{Td}}(r)$) in the region r can be defined as

$$V_{\mathrm{Td}}(r) = V(r) \times N_{\mathrm{NC}}(r) \tag{3.2}$$

where $N_{\mathrm{NC}}(r)$ is the number of non-critical path in the region r and $V(r)$ is the vulnerability of region r as defined by Eq. (3.1).

To stand power-based detection technique, Trojans should be connected to nets with low transition probabilities and placed in regions with enough whitespace and unused routing channel. The value of vulnerability to power-resistant Trojans ($V_{\mathrm{Tp}}(r)$) in the region r can be defined as

$$V_{\mathrm{Tp}}(r) = V(r) \times N_{\mathrm{LP}}(r) \tag{3.3}$$

where $N_{\mathrm{LP}}(r)$ is the number of nets with transition probability smaller than a predefined P_{th} in the region r, and $V(r)$ is the vulnerability of region r as defined by Eq. (3.1).

Trojans resistant to multi-parameter detection techniques should be placed in regions with empty space and unused routing channels and connected to nets with low transition probabilities located on non-critical paths. The value of vulnerability to power and delay-resistant Trojans ($V_{\mathrm{Tdp}}(r)$) in region r can be defined as

$$V_{\mathrm{Tdp}}(r) = V(r) \times N_{\mathrm{NC\&LP}}(r) \tag{3.4}$$

where $N_{\mathrm{NC\&LP}}(r)$ is the number of nets with transition probability less than a predefined P_{th} on non-critical paths in the region r and $V(r)$ is the vulnerability of region r as defined by Eq. (3.1).

3.3.2 Net Analysis

The Net Analysis performs a comprehensive analysis of each net in a circuit. The analysis determines the transition probability of each net in the circuit. With incorporating the circuit layout information, it is possible to obtain the distribution of transition probability across the circuit layout. Using a timing analysis tool, the slack distribution of worst paths passing through a net can be obtained. Nets with low transition probability located on non-critical paths are suitable candidates for Trojan trigger inputs.

In conclusion, the layout-level vulnerability analysis flow identifies regions of a circuit that are more vulnerable to Trojans resistant to delay-based, power-based, and multi-parameter-based detection techniques. Furthermore, the vulnerability of a region is quantified, and this provides a detailed and fair comparison between different circuit implementations. With such knowledge, it is possible to incorporate effective prevention techniques with the least impact on main design specifications. In addition, the flow may provide insightful guidance for authenticating circuits after manufacturing.

b18 benchmark is synthesized using Synopsys's SAED_EDK90nm library at 90nm technology node [17] and 9 metal layers for routing is considered. The layout is divided into tiles with the area A_T equal to W^2 where W is the width of the largest

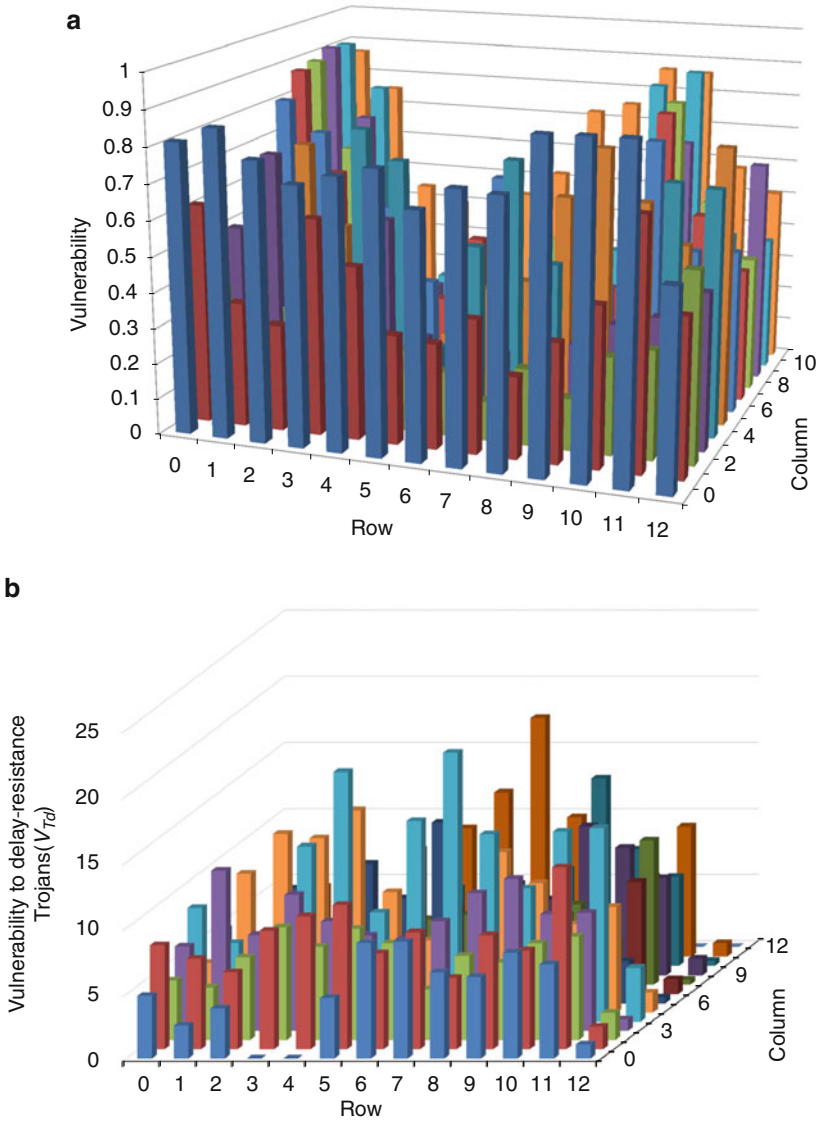


Fig. 3.3 Existence of unused space and routing channels and vulnerability to delay-resistant hardware Trojans in b15 benchmark. (**a**) Unused space and routing channels($V(r)$). (**b**) Delay-resistant Trojans

cell in the design library; that is, $A_T = 560.7424\,\mu m^2$. The average available white space and unused routing channel are about 41 unit of INVX0 and 0.84 per layer, respectively. After performing layout vulnerability analysis flow, Fig. 3.3a presents the vulnerability of b15 benchmark [18] to Trojan insertion at the layout-level as

defined by Eq. (3.1). The average vulnerability is about 0.46 and about 40 % of regions have V above 0.5. These signify considerably high susceptibility of the layout to Trojan insertion.

Figure 3.3b shows the vulnerability of b15 benchmark to Trojans resilient to delay-based detection techniques across the layout. The results indicate the region 95 in Row 7 and Column 4 is the most susceptible region to delay-resistant Trojan with $V_{\mathrm{Td}}(95) = 20.44$, where $N_{\mathrm{NC}}(95) = 28$ and $V(95) = 0.73$. Interestingly, the adjacent region 94 in Row 7 and Column 3 has considerably higher number of non-critical paths ($N_{\mathrm{NC}}(94) = 40$); however, the region 94 has low whitespaces or unused routing channels, $V(94) = 0.21$. Therefore, the susceptibility of region 94 is much lower with $V_{\mathrm{Td}}(94) = 8.4$. The vulnerability of b15 benchmark to Trojans resilient to power-based detection techniques across the layout is shown in Fig. 3.4a with $P_{th} = 1e-04$. The results show that the region 147 in row 11 and column 4 has the maximum vulnerability to power-resistant Trojan with $V_{\mathrm{Tp}}(147) = 7.71$ where $V(147) = 0.70$ and $N_{\mathrm{LP}}(147) = 11$.

Comparing results for delay-resistant Trojans and power-resistant Trojans reveals that b15 benchmark in more susceptible to delay-resistant Trojans as the maximum V_{Td} is greater than the maximum V_{Tp}. Furthermore, regions with the maximum V_{Td} and V_{Tp} are different such that the most susceptible region to delay-resistant Trojan is region 95 and the most susceptible region to power-resistant Trojan is region 147 for b15 benchmark.

Figure 3.4b shows the vulnerability of b15 benchmark to Trojans resilient to multi-parameter power and delay Trojan detection techniques. The results reveal that the region 150 in row 11 and column 7 is the most susceptible region to Trojans resilient to power and delay-based detection techniques with $V_{\mathrm{Tdp}}(150) = 5.32$ where $V(150) = 0.53$ and $N_{\mathrm{NC\&LP}}(150) = 10$. The analysis signifies even with using multi-parameter Trojan detection techniques, it is possible to implement a Trojan whose full activation probability can be about $1 - E40$ while there is still considerable whitespace and unused routing channels in the region 150. This analysis flags regions that are highly susceptible to Trojan insertion; therefore, it can effectively limit Trojan investigation into a limited number of regions. The detailed analysis for b15 benchmark shows that 21 regions out of 169 regions have V_{Tdp} above 3 that indicates the moderate existence of regions with considerable whitespace and unused routing channels and a considerable number of nets with low transition probability on non-critical paths.

The detailed results for b15 benchmark show that the percentage of regions with V_{Td} above 5, 10, and 15 are 75 %, 17 %, and 3 %, respectively. The percentage of regions with V_{Tp} above 2, 4, and 5 are 14 %, 4 %, and 0.6 %, and the percentage of regions with V_{Tdp} above 2, 4, and 5 are 12 %, 2 %, and 0 %. The results emphasize that the b15 benchmark has higher percentage of regions vulnerable to Trojans resilient to delay-based Trojan detection techniques. Further, by using a multi-parameter power and delay Trojan detection techniques these percentages significantly drop.

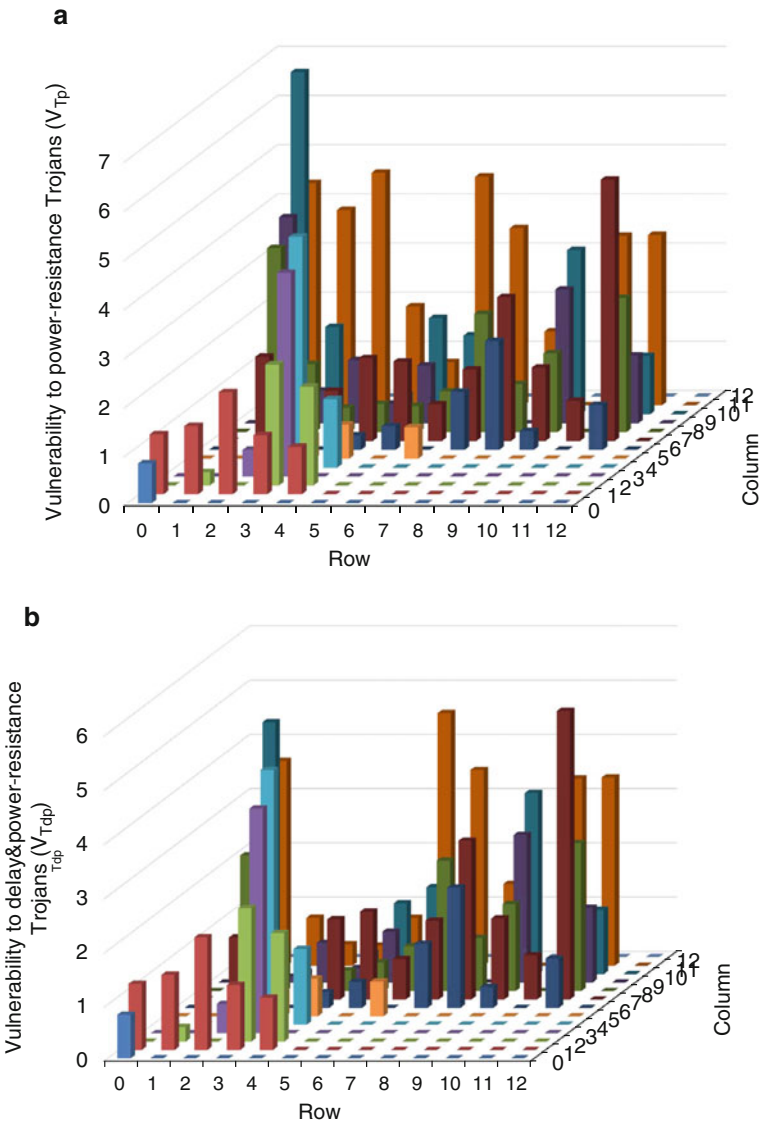


Fig. 3.4 Vulnerability of b15 benchmark to Trojans resilient to power-based and multi-parameter-based detection techniques. (**a**) Power-resilient Trojans with $P_{th} = 1e - 04$. (**b**) Multi-parameter-resilient Trojans

3.4 Trojan Analyses

A Trojan's impact on circuit characteristics depends on its implementation. Trojan inputs tapped from nets with higher transition probabilities will aggrandize switching activity inside the Trojan circuit and increase its contribution to circuit power consumption. Furthermore, the Trojan might affect circuit delay characteristics due to additional capacitance induced by extra routing and Trojan gates. To quantitatively determine the difficulty of detecting a gate-level Trojan, a procedure is developed to determine Trojan detectability based on its impact on delay and power across different circuits. Trojan detectability can establish a fair comparison among different hardware Trojan detection techniques since it is based on induced variations by a Trojan in side-channel signals.

The Trojan detectability metric is determined by (1) the number of transitions in the Trojan circuit and (2) extra capacitance induced by Trojan gates and their routing. This metric is designed to be forward-compatible with new approaches for Trojan detection by introducing a new variable, for example, a quantity related to the electromagnetic field.

Transitions in a Trojan circuit reflect Trojan contribution to circuit power consumption, and Trojan impact on circuit delay characteristic is represented by measuring the added capacitance by the Trojan. Assuming A_{Trojan} represents the number of transitions in the Trojan circuit, S_{Trojan} the Trojan circuit size in terms of the number of cells, A_{TjFree} the number of transitions in the Trojan-free circuit, S_{TjFree} the Trojan-free circuit size in terms of the number of cells, TIC the added capacitance by Trojan as Trojan-induced capacitance, and C_{TjFree} the Trojan-affected path with the largest capacitance in the corresponding Trojan-free circuit, Trojan detectability ($T_{\text{Detectability}}$) at the gate-level is defined as

$$T_{\text{Detectability}} = |t| \tag{3.5}$$

where

$$t = \left(\frac{A_{\text{Trojan}}/S_{\text{Trojan}}}{A_{\text{TjFree}}/S_{\text{TjFree}}}, \frac{\text{TIC}}{C_{\text{TjFree}}} \right) \tag{3.6}$$

$T_{\text{Detectability}}$ at the gate-level is calculated as follows:

1. Apply random inputs to a Trojan-free circuit and obtain the number of transitions in the circuit (A_{TjFree}).
2. Apply the same random vectors to the circuit with a Trojan and obtain the number of transitions in the Trojan circuit (A_{Trojan}).
3. Perform the delay analysis on the Trojan-free and Trojan-inserted circuits.
4. Obtain the list of paths whose capacitance is changed by the Trojan.
5. Determine the Trojan-affected path with the largest capacitance in the corresponding Trojan-free (C_{TjFree}) and the added capacitance (TIC).

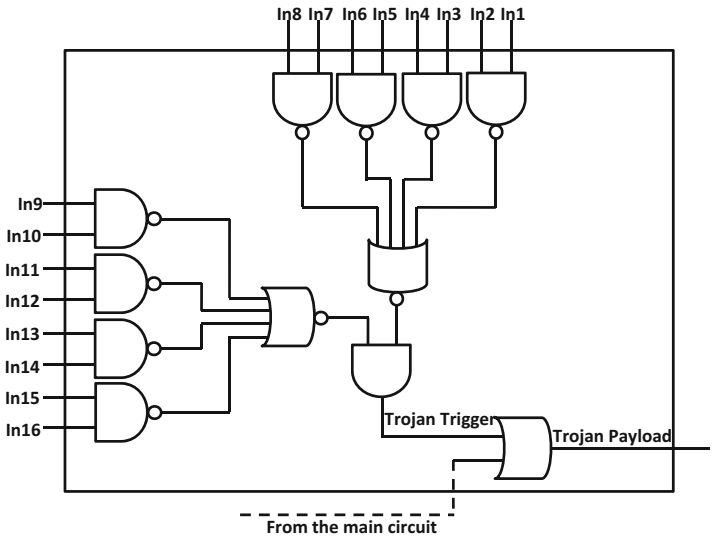


Fig. 3.5 An example comparator Trojan

Table 3.1 The detectability of the comparator Trojan placed at four different locations in Ethernet MAC 10GE circuit

Trojan	A_{TjFree}	S_{TjFree}	A_{Trojan}	S_{Trojan}	TIC (pF)	C_{TjFree} (pF)	$T_{\text{Detectability}}$
TjG-Loc1	106,664,486	102,047	10,682	12	0.000286935	0.041358674	0.851659
TjG-Loc2	106,664,486	102,047	4229	12	0.004969767	0.072111502	0.344132
TjG-Loc3	106,664,486	102,047	3598	12	0.005005983	0.049687761	0.304031
TjG-Loc4	106,664,486	102,047	13,484	12	0.004932996	0.052602269	1.079105

6. Form the vector t (3.6) and compute $T_{\text{Detectability}}$ as defined in Eq. (3.5). Note that Trojan detectability represents the difficulty of detecting a Trojan.

As an example, the comparator Trojan, shown in Fig. 3.5, is inserted at four different locations, namely TjG-Loc1, TjG-Loc2, TjG-Loc3, and TjG-Loc4 (G represents "gate level"), in the Ethernet MAC 10GE circuit, and Table 3.1 shows their detectability. The Ethernet MAC 10GE circuit consists of 102,047 cells, Column 3 S_{TjFree}, while the Trojan size with 12 cells, Column 5 S_{Trojan}, is only about 0.011 % of the entire circuit. TjG-Loc4, in Row 5, experiences the largest switching activity (13,484 in Column 4) and relatively induces high TIC (0.004932996 pF in Column 6). It is expected that TjG-Loc4 will be the easiest Trojan to be detected due to more impact on circuit side-channel signals, and in turn the detectability of TjG-Loc4 ($T_{\text{Detectability}} = 1.079105$ in Column 8) is higher than the others. Although the induced capacitance by TjG-Loc2 (0.004969767 pF), in Row 3, is more than the capacitance induced by TjG-Loc1 (0.000286935 pF), in Row 2, TjG-Loc1 has more significant contribution into circuit switching activity, 10,682 versus 4229 in

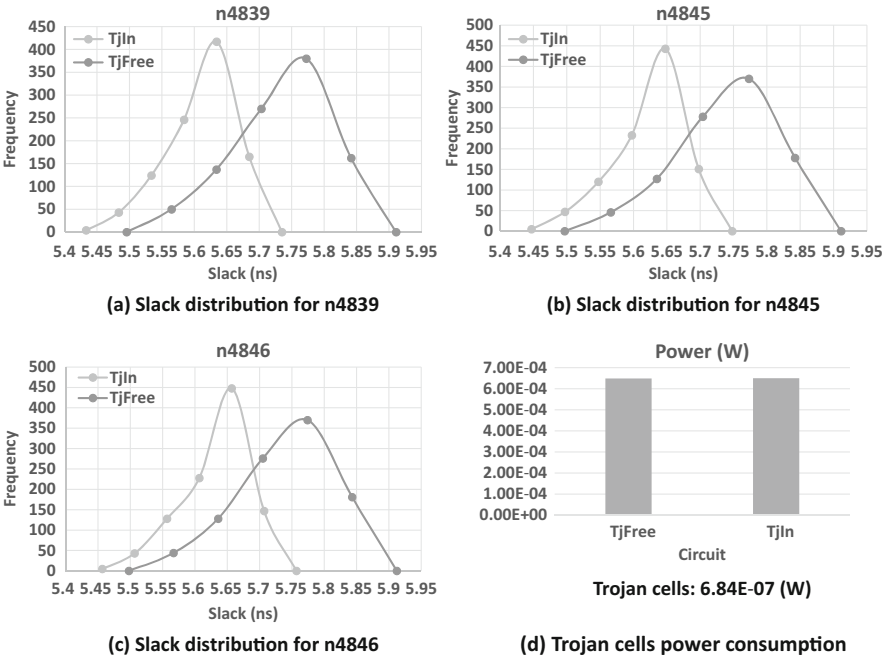


Fig. 3.6 The circuit power consumption and the slack distribution of 1000 worst paths passing through each triggering signal of a 3-bit synchronous counter Trojan inserted in Region 42 of b15 benchmark with $V_{\text{Tdp}}(42) = 4.149$

Column 4. Therefore, TjG-Loc1 has the second largest detectability (0.851659) after TjG-Loc4. Among TjG-Loc2 and TjG-Loc3, although TjG-Loc3, in Row 4, has slightly larger induced capacitance (0.005005983 pF), TjG-Loc2 experiences more switching activity (4229 versus 3598 in Column 4). The two Trojans have close detectability where TjG-Loc2 stands above and TjG-Loc3 remains the hardest Trojan to be detected with the lowest Trojan detectability.

At the layout-level, the V_{Tdp} metric determines the vulnerability of a region to Trojans which are resistant to both delay-based detection techniques and power-based detection techniques. One 3-bit synchronous counter Trojan and one 12-bit comparator separately inserted into b15 benchmark and Figs. 3.6 and 3.7, respectively, show their delay and power impacts. The counter Trojan is inserted in the region 42 at Column 3 and Row 3 with $V_{\text{Tdp}}(42) = 4.149$ ($V(42) = 0.4149$ and $N_{\text{NC\&LP}}(42) = 10$). Figure 3.6a–c shows the slack distribution of the 1000 worst paths passing through the three triggering signals of the counter. The analysis indicates the amount of TID for the three signals is about $1ns$, on average, and the minimum slack for each signal after Trojan insertion still is so large that the worst path is not become a critical path. Figure 3.6d also presents the very small power consumption of the Trojan circuit ($\approx 6.84E-07$ W), and the circuit power consumption before and after Trojan insertion is almost remained the same, about

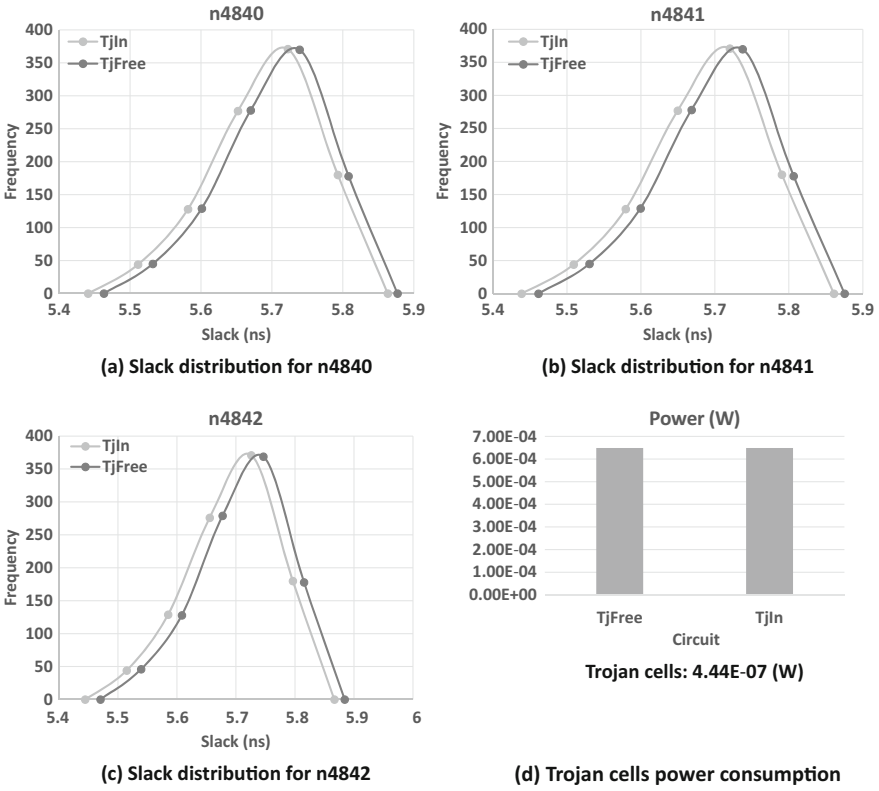


Fig. 3.7 The circuit power consumption and the slack distribution of 1000 worst paths passing through three selected triggering signal of a 12-bit comparator Trojan with minimum slack inserted in Region 43 of b15 benchmark with $V_{\text{Tdp}}(43) = 4.7035$

$6.49E-04$ W. Therefore, the 3-bit synchronous counter Trojan may remain hidden from both delay- and power-based Trojan detection techniques.

A similar analysis is performed for 12-bit comparator inserted in the neighboring region 43 at Column 4 and Row 3 with $V_{\text{Tdp}}(43) = 4.7035$ ($V(43) = 0.78$ and $N_{\text{NC\&LP}}(43) = 6$). The slack distributions of three selected inputs of the comparator with the minimum slacks are presented in Fig. 3.7a–c. The amount of TID is 0.018 ns, on average, and the delay of the worst path is not large enough to be considered a critical path. In Fig. 3.7d, the Trojan power consumption is very small ($\approx 4.44E-07$ W). With small impact on circuit delay characteristics and power consumption, the comparator Trojan may also remain hidden. Comparing the 3-bit synchronous counter to 12-bit combinational comparator indicate that the counter consumes more power than the comparator although the size of comparator circuit larger. This is attributed to the fact that the counter is a sequential circuit and the clock inputs of its flip-flops are connected to circuit clock. Furthermore, TID for the

comparator circuit is smaller than that of the counter this is because of the higher V_{Tdp} value of the comparator. Therefore, the V_{Tdp} metric can effectively identify regions vulnerable to Trojans resilient to delay&power-based techniques.

3.5 Conclusions

This chapter presented a novel gate- and layout-level vulnerability analysis flows presented to determine susceptibility of a gate-level netlist and a circuit layout to hardware Trojan insertion. Based on a circuit topology and placement and routing information, several metrics were defined to quantify the vulnerability of circuit layout to different types of Trojans. The significance of introduced metrics was evaluated by implementing different Trojans. The results indicated the considerable vulnerability of gate-level netlist and circuit layouts to Trojans resistant to delay-based, power-based, and multi-parameter-based Trojan detection techniques. The proposed novel layout vulnerability analysis flow may provide guidance for Trojan prevention during circuit development and Trojan detection after fabrication.

References

1. U.S.D. Of Defense, Defense science board task force on high performance microchip supply (2015) http://www.acq.osd.mil/dsb/reports/2005-02-HPMS_Report_Final.pdf
2. S. Adee, The hunt for the kill switch (2008) http://www.spectrum.ieee.org/print/6171
3. S. Bhunia, M. Abramovici, D. Agarwal, P. Bradley, M.S. Hsiao, J. Plusquellic, M. Tehranipoor, Protection against hardware Trojan attacks: towards a comprehensive solution. IEEE Des. Test **30**(3), 6–17 (2013)
4. M. Tehranipoor, F. Koushanfar, A survey of hardware Trojan taxonomy and detection. IEEE Des. Test Comput. **27**(1), 10–25 (2010)
5. R. Karri, J. Rajendran, K. Rosenfeld, M. Tehranipoor, Trustworthy hardware: identifying and classifying hardware Trojans. IEEE Comput. **43**(10), 39–46 (2010)
6. M. Tehranipoor, H. Salmani, X. Zhang, X. Wang, R. Karri, J. Rajendran, K. Rosenfeld, Trustworthy hardware: Trojan detection and design-for-trust challenges. IEEE Comput. **44**(7), 66–74 (2011)
7. Y. Jin, D. Maliuk, Y. Makris, Post-deployment trust evaluation in wireless cryptographic ICs, in *Proceedings of the IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE12)* (2012), PP. 965–970
8. X. Zhang, M. Tehranipoor, Case study: detecting hardware Trojans in third-party digital IP cores, in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust (HOST11)* (2011), pp. 67–70
9. Y. Jin, Y. Makris, Hardware Trojan detection using path delay fingerprint, in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust (HOST08)* (2008), pp. 51–57
10. J. Li, J. Lach, At-speed delay characterization for IC authentication and Trojan horse detection, in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust (HOST08)* (2008), pp. 8–14
11. X. Wang, H. Salmani, M. Tehranipoor, J. Plusquellic, Hardware Trojan detection and isolation using current integration and localized current analysis, in *Proceedings of the IEEE International Symposium on Fault and Defect Tolerance in VLSI Systems (DFT08)* (2008), pp. 87–95

12. D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, B. Sunar, Trojan detection using IC fingerprinting, in *Proceedings of the IEEE Symposium on Security and Privacy* (2007), pp. 296–310
13. R. Rad, X. Wang, J. Plusquellic, M. Tehranipoor, Power supply signal calibration techniques for improving detection resolution to hardware Trojans, in *Proceedings of the International Conference on Computer-Aided Design(ICCAD08)* (2008), pp. 632–639
14. M. Banga, M.S. Hsiao, A novel sustained vector technique for the detection of hardware Trojans, in *Proceedings of the International Conference on VLSI Design (VLSID09)* (2009), pp. 327–332
15. F. Wolff, C. Papachristou, S. Bhunia, R.S. Chakraborty, Towards Trojan-free trusted ICs: problem analysis and detection scheme, in *Proceedings of the Design, Automation and Test in Europe (DATE08)* (2008), pp. 1362–1365
16. Ethernet 10GE MAC (2013) http://opencores.org/project,xge_mac
17. Synopsys 90nm generic library for teaching IC design (2016) http://www.synopsys.com/Community/UniversityProgram/Pages
18. ISCAS benchmarks (2016) http://www.pld.ttu.ee/~maksim/benchmarks/

Chapter 4
Code Coverage Analysis for IP Trust Verification

Adib Nahiyan and Mark Tehranipoor

4.1 Introduction

Due to globalization of the semiconductor design and fabrication process, integrated circuits (ICs) are becoming increasingly vulnerable to malicious activities and alterations. Today's system-on-chips (SoCs) usually contain tens of IP cores (digital and analog) performing various functions. In practice, very seldom IPs are developed by the SoC integrator; in fact most of them are currently being designed offshore by 3PIP vendors. This raises a major concern towards the trustworthiness of 3PIPs. These concerns have been documented in various reports and technical events [1].

IP trust problem is defined as the possibility of inserting Trojan into 3PIPs by IP vendors during SoC design. This issue has gained significant attention as a Trojan-inserted by 3PIP vendor can create backdoors in the design through which sensitive information can be leaked and other possible attacks (e.g., denial of service, reduction in reliability, etc.) can be performed [2, 3].

Detection of Trojans in 3PIP cores is extremely difficult as there is no golden version against which to compare a given IP core during verification. In theory, an effective way to detect a Trojan in an IP core is to activate the Trojan and observe its effects, but the Trojan's type, size, and location are unknown, and its activation condition is most likely a rare event [4]. The conventional side-channel techniques for IC trust are not applicable to IP trust. When a Trojan exists in an IP core, all the fabricated ICs will contain Trojans. The only trusted component would be the specification from the SoC designer which defines the function, primary input and output, and other information about the 3PIP that they intend to use in their systems. A Trojan can be very well hidden during the normal functional operation of the 3PIP

A. Nahiyan (✉) • M. Tehranipoor
University of Florida, Gainesville, FL, USA
e-mail: adib1991@ufl.edu; tehranipoor@ece.ufl.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_4

supplied as register transfer level (RTL) code. A large industrial-scale IP core can include thousands of lines of code. Identifying the few lines of RTL code in an IP core that represents a Trojan is an extremely challenging task.

Given the complexity of the problem, there is no silver bullet available to detect hardware Trojans in 3PIPs. An IP trust verification technique that tries to address the IP trust problem is presented in [5]. This chapter will present this technique in details. Several concepts such as formal verification, code coverage analysis, and ATPG methods have been employed in this technique to achieve high confidence in whether the circuit is Trojan-free or Trojan-inserted. This IP trust verification technique is based on identification of suspicious signals. Suspicious signals are identified first by coverage analysis with improved test bench. Removing redundant circuit and equivalence theorems are then applied to reduce the number of suspicious signals. Sequential ATPG is used to generate patterns to activate suspicious signals which can trigger the Trojan.

4.2 SoC Design Flow

A typically SoC design flow is shown in Fig. 4.1. Design specification by the SoC integrator is generally the first step. The SoC integrator then identifies a list of IPs necessary to implement the given specification. These IP cores are either developed in-house or purchased from 3PIP vendors. These 3PIP cores can be procured from the vendors in one of the following three ways [6]:

- soft IP cores are delivered as synthesizable register transfer level (RTL) hardware description language (HDL)
- hard IP cores are delivered as GDSII representations of a fully placed and routed core design
- firm IP cores are optimized in structure and topology for performance and area, possibly using a generic library.

After developing/procuring all the necessary soft IPs, the SoC design house integrates them to generate the RTL specification of the whole system. SoC integrator then synthesizes the RTL description into a gate-level netlist based on the logic cells and I/Os of a target technology library, then they may integrate gate-level IP cores from a vendor into this netlist. They also add design-for-test (DFT) structures to improve the design's testability. The next step is to translate the gate-level netlist into a physical layout based on logic cells and I/O geometries. It is also possible to import IP cores from vendors in GDSII layout file format. After performing static timing analysis (STA) and power closure, developers generate the final layout in GDSII format and send it out for fabrication.

Today's advanced semiconductor technology requires prohibitive investment for each stage of the SoC development procedure. As an example, the estimated cost of owning a foundry was five billion dollar in 2015 [7]. As a result, most semiconductor companies cannot afford maintaining such a long supply chain from

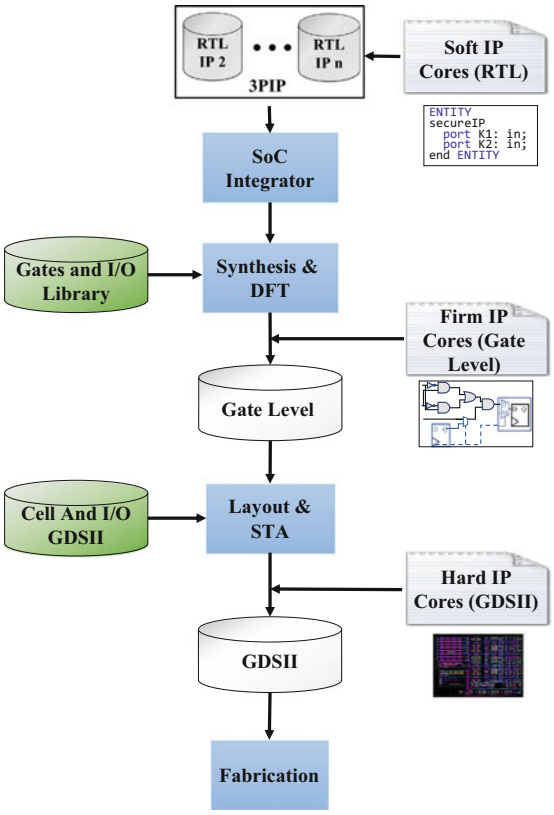


Fig. 4.1 System-on-chip (SoC) design flow consists of IP-core-based design, system integration, and manufacturing

design to packaging. In order to lower R&D cost and speed up the development cycle, the SoC design houses typically outsource fabrication to a third-party foundry, purchase third-party intellectual property (IP) cores, and/or use Electronic Design Automation (EDA) tools from third-party vendors. The use of untrusted (and potentially malicious) third parties increases the security concerns. Thus, the supply chain is now considered susceptible to various attacks, such as hardware Trojan insertion, reverse engineering, IP piracy, IC tampering, IC cloning, IC overproduction, and so forth. Among these, hardware Trojans are arguably one of the biggest concerns and have garnered considerable attention [9].

Trojans can be inserted in SoCs at the RTL, at the gate level during synthesis and DFT insertion, at the layout level during placement and routing, or during IC manufacturing [8]. An attacker can also insert a Trojan through IP cores provided by external vendors. Designers must verify the trustworthiness of IP cores to ensure that they perform as intended, nothing more and nothing less.

4.3 Hardware Trojan Structure

A hardware Trojan is defined as a malicious, intentional modification of a circuit design that results in undesired behavior when the circuit is deployed [8]. The basic structure of a Trojan in a 3PIP can include two main parts, trigger and payload. A Trojan trigger is an optional part that monitors various signals and/or a series of events in the circuit. The payload usually taps signals from the original (Trojan-free) circuit and the output of the trigger. Once the trigger detects an expected event or condition, the payload is activated to perform malicious behavior. Typically, the trigger is expected to be activated under extremely rare conditions, so the payload remains inactive most of the time. When the payload is inactive, the IC acts like a Trojan-free circuit, making it difficult to detect the Trojan.

Figure 4.2 shows the basic structure of the Trojan at gate level. The trigger inputs ($T_1, T_2, \ldots, T_k$) come from various nets in the circuit. The payload taps the original signal Net_i from the original (Trojan-free) circuit and the output of the trigger. Since the trigger is expected to be activated under rare condition, the payload output stays at the same value as Net_i most of the time. However, when trigger is active (i.e., *TriggerEnable* is "0") the payload output will be different from Net_i; this could result in injecting an erroneous value into the circuit and causing error at the output. Note that Trojans at RTL would have similar functionality to the one shown in Fig. 4.2.

4.4 Related Work

IP trust validation focuses on verifying that an IP does not perform any malicious function, i.e., an IP does not contain any Trojan. Existing IP trust validation techniques can be broadly classified into code/structural analysis, functional verification, logic testing, formal verification, and runtime validation.

Code Coverage Analysis Code coverage is defined as the percentage of lines of code that has been executed during functional verification of the design. This metrics

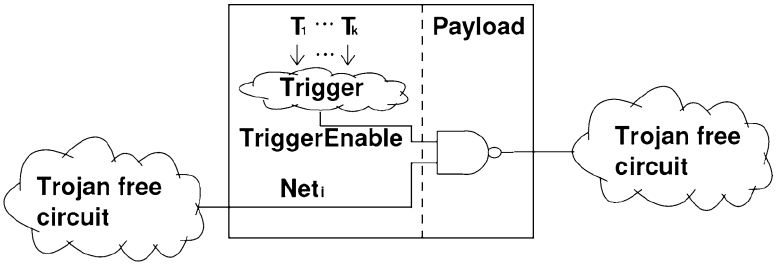


Fig. 4.2 Trojan structure

gives a quantitative measure of the completeness of the functional simulation of the design. Code coverage analysis can also be applied to identify suspicious signals that may be a part of a Trojan and validate the trustworthiness of a 3PIP. In [9], authors have proposed a technique named unused circuit identification (UCI) to find the lines of RTL code that have not been executed during simulation. These unused lines of codes can be considered to be part of a malicious circuit. Authors in [9] proposed to remove these suspicious lines of RTL code from the hardware design and emulate it in the software level. In [10], authors have proposed similar code coverage analysis in combination with hardware assertion checker to identify malicious circuitry in a 3PIP. However, these techniques do not guarantee the trustworthiness of a 3PIP. Authors in [11] have demonstrated that hardware Trojans can be designed to defeat UCI technique. This type of Trojans derives their triggering circuits from less likely events to evade detection from code coverage analysis.

Formal Verification Formal methods such as symbolic execution [12], model checking [13], and information flow [14] have been traditionally applied to software systems for finding security bugs and improving test coverage. Formal verification has also shown to be effective in verifying the trustworthiness of 3PIP [15–17]. These approaches are based on the concept of proof-carrying code (PCC) to formally validate the security-related properties of an IP. In these proposed approaches, an SoC integrator provides a set of security properties in addition to the standard functional specification to the IP vendor. A formal proof of these properties alongside with the hardware IP is then provided by the third-party vendor. SoC integrator then validates the proof by using the PCC. Any malicious modification of the IP would violate this proof indicating the presence of hardware Trojan. However, these approaches cannot ensure complete trust in an IP because the third-party vendor crafts the formal proof of these security-related properties [18].

Authors in [19] have proposed a technique to formally verify malicious modification of critical data in 3PIP by hardware Trojans. The proposed technique is based on bounded model checking (BMC). Here, the BMC checks for the property—"does critical information get corrupted?" and outputs if the property is being violated in the given IP. Also BMC reports the sequence of input patterns which violates this property. Form the reported input patterns it is possible to extract the triggering condition of the Trojan. Another similar approach has been proposed in [20] which formally verifies unauthorized information leakage in 3PIPs. This technique checks for the property—"does the design leak any sensitive information?" The limitation of these approaches is that the processing ability of the model checking is relatively limited, due to the problem of space explosion.

Structural Analysis Structural analysis employs quantitative metrics to mark signals or gates with low activation probability as suspicious. In [21], authors have presented a metric named "Statement Hardness" to evaluate the difficulty of executing a statement in the RTL code. Areas in a circuit with large value of "Statement Hardness" are more vulnerable to Trojan insertion. At gate level, an attacker would most likely target *hard-to-detect* areas of the gate-level netlist to

insert Trojan. *Hard-to-detect* nets are defined as nets which have low transition probability and are not testable through well-known fault testing techniques (stuck-at, transition delay, path delay, and bridging faults) [22]. Inserting a Trojan in *hard-to-detect* areas would reduce the probability to trigger the Trojan and thereby, reduce the probability of being detected during verification and validation testing. In [23], authors have proposed metrics to evaluate *hard-to-detect* areas in the gate-level netlist. The limitations of code/structural analysis techniques are that they do not guarantee Trojan detection, and manual post-processing is required to analyze suspicious signals or gates and determine if they are a part of a Trojan.

Logic Testing Logic testing aims to activate Trojans by applying test vectors and comparing the responses with the correct results. While at first glance this is similar in spirit to manufacturing tests for detecting manufacturing defects, conventional manufacturing tests using functional/structural/random patterns perform poorly to reliably detect hardware Trojans [24]. Intelligent adversaries can design Trojans that are activated under very rare conditions, so they can go undetected under structural and functional tests during the manufacturing test process. In [25] authors have developed a test pattern generation methods to trigger such rarely activated nets and improve the possibility of observing Trojan's effects from primary outputs. However, this technique does not guarantee to trigger the Trojan and it is infeasible to apply this technique in an industrial-scale design. Additionally, there could be Trojans which do not functionally affect the circuit but leaks confidential information through side channel. This type of Trojans cannot be identified through logic testing.

Functional Analysis Functional analysis applies random input patterns and performs functional simulation of the IP to find suspicious regions of the IP which have similar characteristics of a hardware Trojan. The basic difference between functional analysis and logic testing is that logic testing aims to apply specific patterns to activate a Trojan, whereas functional analysis applies random patterns and these patterns are not directed to trigger the Trojan. Authors in [26] have proposed a technique named Functional Analysis for Nearly unused Circuit Identification (FANCI) which flags nets having weak input-to-output dependency as suspicious. This approach is based on the observation that a hardware Trojan is triggered under very rare condition. Therefore the logic implementing the trigger circuit of a Trojan is nearly unused or dormant during normal functional operation. Here, the authors have proposed a metric called "Control value" to find "nearly unused logic" by quantifying the degree of controllability of each input net has on its output function. "Control value" is computed by applying random input patterns and measuring the number of output transitions. If the control value of a net is lower than a predefined threshold, then the net is flagged as suspicious. For example, for the RSA-T100 (http://trust-hub.org/resources/benchmarks) Trojan, the triggering condition is *32'h44444444*. The "Control value" for the triggering net is 2^{-32} which is expected to be lower than the predefined threshold. The major limitations of FANCI are, this approach produces a large number of false positive results and this approach does not specify any method to verify if the suspicious

signals are performing any malicious operation. Also, authors in [27] have shown to design Trojans to defeat FANCI. Here, they design Trojan circuits whose trigger vector arrives over multiple clock cycles. For example, for the RSA-T100 Trojan, the triggering sequence can be derived over four cycles making the "Control value" of the triggering net 2^{-8}. Furthermore, FANCI cannot identify "Always On" Trojans which remains active during their lifetime and do not have any triggering circuitry.

Authors in [28] have proposed a technique called VeriTrust to identify potential triggering inputs of a hardware Trojan. The proposed technique is based on the observation that input ports of the triggering circuit of a hardware Trojan keep dormant during normal operation and therefore, are redundant to the normal logic function of the circuit. VeriTrust works as follows, first it performs functional simulation of the IP with random input patterns and traces the activation history of the inputs ports in the form of sums-of-products (SOP) and product-of-sums (POS). Veritrust then identifies redundant inputs by analyzing the SOPs and POSs which are unactivated during functional simulation. These redundant input signals are potential triggering inputs of a hardware Trojan. VeriTrust technique aims to be independent of the implementation style of hardware Trojan. However, this technique also produces a large number of false positive results because of incomplete functional simulation and unactivated entries belonging to normal function. Also, authors in [27] have designed Trojans which can defeat VeriTrust by ensuring that Trojan triggering circuit is driven by a subset of functional inputs in addition to triggering inputs. VeriTrust also shares the same limitation of FANCI as not being able to identify "Always On" Trojans.

Runtime Validation Runtime validation approaches are generally based on dual modular redundancy-based approaches [29]. These techniques rely on procuring IP cores of same functionality from different IP vendors. The basic assumption is that, it is highly unlikely that different Trojans in different 3PIPs will produce identical wrong outputs. Therefore, by comparing the outputs of IP cores of same functionality but obtained from different vendors, one can detect any malicious activity triggered by a Trojan. The main disadvantage of this approach is that the area overhead is prohibitively high ($\tilde{1}00\,\%$ area overhead) and the SoC integrator needs to purchase same functional IP from different vendors (economically infeasible).

4.5 A Case Study for IP Trust Verification

It is evident from the discussion on Sect. 4.4 that it is not possible to verify with high confidence the trustworthiness of a 3PIP using one single approach. Authors in [5] have proposed an IP trust validation technique which involves formal verification, coverage analysis, redundant circuit removal, sequential ATPG, and equivalence theorems. This technique first utilizes formal verification and code coverage analysis to identify any suspicious signals and components corresponding to Trojans in a

3PIP core. Next, the proposed technique applies suspicious signal analysis to reduce the number of false positive suspicious signals. This technique focuses on ensuring trust in soft IP cores, as they are the most dominant cores in the market today due to the flexibility they offer to the SoC designers.

4.5.1 Formal Verification and Coverage Analysis

One of the important concepts in this proposed technique is formal verification, an algorithmic-based approach to logic verification that exhaustively proves the functional properties of a design. It contains three types of verification methods that are not commonly used in the traditional verification, namely model checking, equivalence checking, and property checking. All functions in the specification are defined as properties. The specific corner cases in the test suite as they monitor particular objects in a 3PIP could also be represented by properties, such as worry cases, inter-block interfaces, and complex RTL structures. They can be represented as properties wherever the protocols may be misused, assumptions violated, or design intent incorrectly implemented. Formal verification uses property checking to check whether the IP satisfies those properties. With property checking, every corner of the design can be explored. For example, in benchmark RS232, there are two main functionalities in the specification: (1) transmitter and (2) receiver. Figure 4.3 shows the waveform of the transmitter. Take the *startbit* as an example; with $Rst == 1'b1$, *clk* positive edge, and $xmitH == 1'b1$, the output signal *Uart_xmit* will start to transmit *startbit* "0". This functionality is described using the SystemVerilog property shown in Fig. 4.4, and the corresponding assertion is defined simultaneously. The remaining items in the specification are also translated to properties during formal verification. Once all the functionalities in the specification are translated to properties, coverage metrics can help identify suspicious parts in the 3PIP under authentication. Those suspicious parts may be Trojans (or part of Trojans).

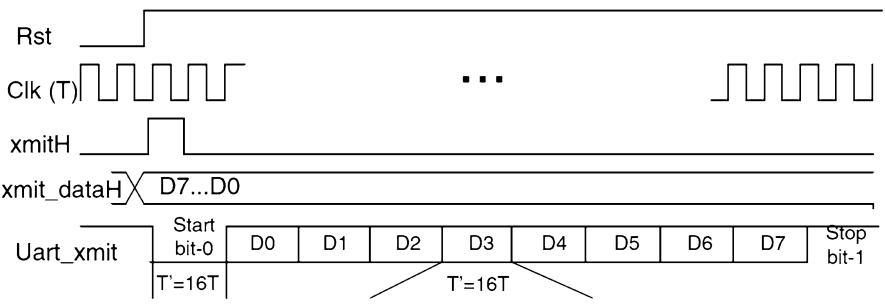


Fig. 4.3 Transmitter property in the specification stage

```
01:   property e1;
02:     @(posedge uart_clk) disable iff (Rst)
03:     $rose(xmitH) |− > ##1 (uart_XMIT_dataH==0);
04:     endproperty
05:
06:     a1: assert property( e1 );
```

Fig. 4.4 One of the properties and assertions definitions for RS232

01: Line No	Coverage	Block Type
02: 69	1	ALWAYS
03: 70	1	CASEITEM
04: 71	1	CASEITEM
05: 72	0	CASEITEM
06: 73	1	CASEITEM
07: 74	0	CASEITEM
08: 82	1	ALWAYS
09: 82.1	1	IF
... ...	...	...

Fig. 4.5 Part of the line coverage report

Coverage metrics include code coverage and functional coverage. Code coverage analysis is a metric that evaluates the effectiveness of a test bench in exercising the design [30, 31]. There are many different types of code coverage analysis, but only a few of them are helpful for IP trust, namely line, statement, toggle, and finite state machine (FSM) coverage. Toggle coverage reports whether signals switch in the gate-level netlist while the other three coverage metrics show which line(s) and statement(s) are executed, and whether states in FSM are reached in RTL code during verification. Figure 4.5 shows parts of line coverage report during the simulation with RS232. This report shows that lines 72 and 74 are not executed, which helps improve the test bench by checking the source code. If the RTL code is easily readable, special patterns that can activate those lines will be added to the test bench. Otherwise, random patterns will be added to verify the 3PIP.

Functional coverage is the determination of how much functionality of the design has been exercised by the verification environment. The functional requirements are imposed on both the design inputs and outputs and on their interrelationships by the design specifications from SoC designer (i.e., IP buyers). All the functional requirements can be translated as different types of assertion, as in Fig. 4.4. Functional coverage checks those assertions to see whether they are successful or not. Table 4.1 shows part of the assertions coverage report (Assertion a1 is defined in Fig. 4.4). The number of attempts in the table means that there are 500,003 positive edge clocks during the simulation time when the tool tries to check the assertion.

Table 4.1 Part of the assertion report with RS232

Assertion	Attempts	Real success	Failure	Incomplete
test.uart1.uart_checker.a1	500,003	1953	0	0
test.uart1.uart_checker.a2	1953	1952	0	1
...	...	...	...	...

In Table 4.1, the Real Success column represents the assertion success rate while Failure/Incomplete column denotes the frequency of assertion failure/incomplete. When there are zero Failures, the property is always satisfied.

If all the assertions generated from the specification of the 3PIP are successful and all the coverage metrics such as line, statement, and FSM are 100 %, then it can be assumed with high confidence that the 3PIP is Trojan-free. Otherwise, the uncovered lines, statements, states in FSM, and signals are considered suspicious. All the suspicious parts constitute the suspicious list.

4.5.2 Techniques for Suspicious Signals Reduction

Based on the formal verification and coverage metric, a flow is proposed to verify the trustworthiness of 3PIP in [1]. The basic idea of the proposed solution is that without redundant circuit and Trojans in a 3PIP, all the signals/components are expected to change their states during verification and 3PIP should function perfectly. Thus, the signals/components that stay stable during toggle coverage analysis are considered suspicious, as Trojan circuits do not change their states frequently. Each suspicious signal is then considered as the *TriggercEnablex*. Figure 4.6 shows the flow to identify and minimize the suspicious parts, including test pattern generation, suspicious signal identification, and suspicious signal analysis. Each step in the figure will be discussed in detail in the following.

4.5.2.1 Phase 1: Test Bench Generation and Suspicious Signal Identification

In order to verify the trustworthiness of 3PIPs, 100 % coverage of the test bench is best. However, it is very difficult to achieve 100 % coverage for every 3PIP, especially those with tens of thousands of lines of code. In the flow, the first step is to improve the test bench to obtain a higher code coverage with an acceptable simulation runtime. With each property in the specification and basic functional test vectors, formal verification reports line, statement, and FSM coverage for the RTL code. If one of the assertions fails even once during verification, the 3PIP is considered untrustworthy, containing Trojans or bugs. If all the assertions are successful and the code coverage is 100 %, the 3PIP can be trusted. If at least one

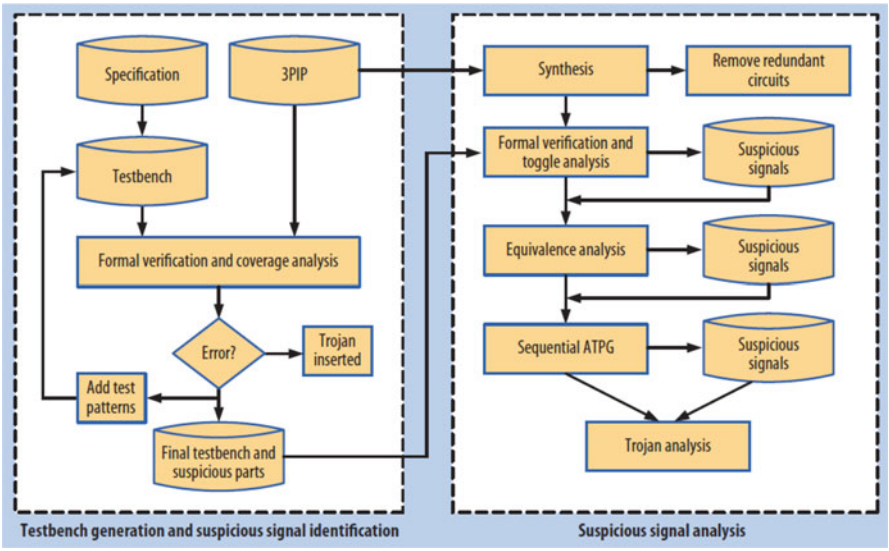


Fig. 4.6 The proposed flow for identifying and minimizing suspicious signals

assertion fails or the code coverage is less than 100 %, more test vectors need to be added to the test bench. The basic purpose of adding new vectors is to activate the uncovered parts as much as possible. But the verification time will increase as the number of test vectors increases. With the acceptable verification time and certain coverage percentage, both defined by the IP buyer, the final test bench will be generated and the RTL source code will be synthesized for further analysis.

4.5.2.2 Phase 2: Suspicious Signals Analysis

Redundant Circuit Removal (RCR) Redundant circuits must be removed from the suspicious list since they also tend to stay at the same logic value during verification, and input patterns cannot activate them. Removing a redundant circuit involves sequential reasoning, SAT-sweeping, conflict analysis, and data mining. The SAT method integrated in the Synopsys Design Compiler (DC) is used in this flow.

Another method to remove redundant circuits is developed in [5]. Scan chains are inserted into the gate-level netlist after synthesis for design testability, and ATPG generates patterns for all the stuck-at faults. The untestable stuck-at faults during ATPG are likely to be redundant logic. The reason is that if the stuck-at fault is untestable, the output responses of the faulty circuit will be identical to the output of the fault-free circuit for all possible input patterns. Thus, when ATPG identifies a stuck-at-1/0 fault as untestable, the faulty net can be replaced by logic 1/0 in the

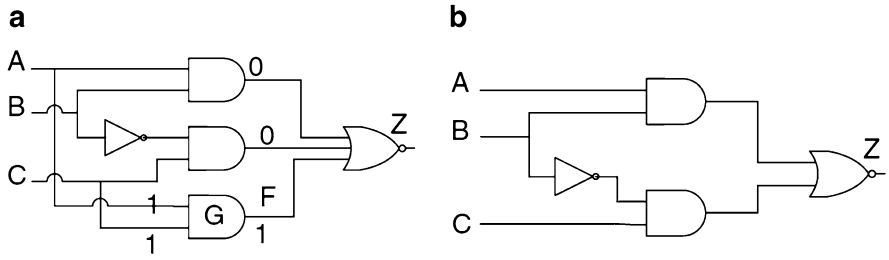


Fig. 4.7 (**a**) Before removing the redundant circuit with untestable F stuck-at-0 fault and (**b**) After removing the redundant circuit

gate-level netlist without scan-chain. All the circuits driving the faulty net will be removed as well. Figure 4.7a shows the circuit before redundant circuit removal.

The stuck-at-0 fault of net F is untestable when generating patterns. Net F will be replaced by 0 and the gate G driving it will be removed from the original circuit, as shown in Fig. 4.7b.

After redundant circuit removal, toggle coverage analysis for the gate-level netlist without scan chain will identify which signals do not toggle (also called quiet signal) during verification with the test bench generated in Phase 1. These signals will be considered suspicious and added to the suspicious list. By monitoring these suspicious signals during verification, the authors obtain the logic value those signal are stuck at.

Equivalence Analysis Fault equivalence theorems are known to reduce the number of faults during ATPG [32]. Similarly, the authors develop suspicious signal equivalence theorems to reduce the number of suspicious signals in [5].

- **Theorem 1.** *If signal A is the D pin of a flip-flop (FF) while signal B is the Q pin of the same FF, the quiet signal A makes signal B quiet. Thus signal A is considered equal to B, which means if the pattern that can activate A is found, it will activate B as well. Then signal B will be removed from the suspicious signal list. As the QN port of an FF is the inversion of the Q port, they will stay quiet or switch at the same time. Thus the suspicious signal B would be considered equal to A and should be removed from the suspicious list.*
- **Theorem 2.** *If signal A is the output pin of an inverter while signal B is its input, they will stay quiet or switch at the same time. Thus the suspicious signal B would be considered equal to A and should be removed from the suspicious list.*
- **Theorem 3.** *One of the inputs of AND gate A stuck-at-0 will cause the output B to stay quiet and one of the inputs of OR gate C stuck-at-1 will make the output D high all along. Thus, for AND gate, B stuck-at-0 is identical to A stuck-at-0, while for OR gate, D is identical to C stuck-at-1.*

Sequential ATPG After reducing the number of suspicious signals by applying the above equivalence theorems, the authors use sequential ATPG to generate special patterns to change the value of certain signals during simulation in [5]. Stuck-at

faults are targeted by the sequential ATPG to generate a sequential pattern to activate the suspicious signals when applied to the 3PIP. If the 3PIP functions perfectly with this pattern, the activated suspicious signals are considered part of the original circuit. Otherwise, there must be malicious inclusion in the 3PIP.

4.6 Simulation Results

The flow is applied to the RS232 circuit. Nine Trojans designed by authors in [5] and ten Trojans from (http://trust-hub.org/resources/benchmarks) are inserted into the 3PIP (RS232). In total, there are 19 RS232 benchmarks with one Trojan in each IP. The following section presents the simulation setup and test bench analysis for the 19 Trojan-inserted benchmarks. Next, the results of redundant circuit removal and the reduction of suspicious signals will be presented. Finally, Trojan coverage analysis will be discussed.

4.6.1 Benchmark Setup

The specification, structure, and functionality of the Trojans designed by authors in [5] are discussed below.

Trojan 1 The trigger of Trojan 1 is a special input sequence *8'ha6–8'h75–8'hc0–8'hff* . The payload changes the FSM in the transmitter of RS232 from state *Start* to *Stop*, which means that once the Trojan is triggered, RS232 will stop transmitting data ($outputdata = 8'h0$). Since the trigger of the Trojan is a sequence of four special inputs, the probability of detecting the Trojan during verification is $1/2^{32}$. If the baud rate is 2400 and RS232 transmits 240 words in one second, it will take 207.2 days to activate the Trojan and detect the error. In other words, it would be practically impossible to detect it by conventional verification. When this Trojan is inserted into RS232, an FSM is used to describe the Trojan input sequence. A three-bit variable state represents the FSM.

Trojan 2 This Trojan only adds four lines to the original RTL code. If the transmitting word is odd and the receiving word is *8'haa*, RS232 will stop receiving words. This Trojan is less complex compared to Trojan 1, however, it provides opportunities to demonstrate the effectiveness of each step of the proposed flow.

Trojan 3 The trigger of Trojan 3 is the same as that of Trojan 1, but the payload is different. Trojan 1 changes the state machine while Trojan 3 changes the shift process. The eighth bit of the transmitting word will be replaced by a Trojan bit during transmission. The Trojan bit could be authentication information, the special key to enable the system, or other important information.

Trojan 4 Trojan 4 is designed to act like a time bomb. A counter is inserted into RS232 to count the number of words that have been sent out. After sending *10'h3ff* words, the Trojan will be activated. The sixth bit of the transmitting word will be replaced by a Trojan bit.

Trojan 5 After *24'hffffff* positive edge clock, this Trojan's enable signal will become high. The sixth bit of the transmitting word will be replaced by a Trojan bit.

Trojan 6 If RS232 receives "0" when the system is reset, the Trojan will be activated. The eighth bit of the transmitting word will be replaced by a Trojan bit.

Trojan 7 When the transmitter sends a word *8'h01* and the receiver receives a word *8'hef* at the same time, the Trojan will be activated. A Trojan bit will replace the first bit of the transmitting word.

Trojans 8 and 9 These Trojans do not tamper the original function of RS232 but add extra one stage (Trojan 8) and three stage (Trojan 9) ring oscillator to the RTL, which will increase the temperature of the chip quickly if they get activated.

4.6.2 Impact of Test Bench on Coverage Analysis

All the items in the specification are translated into properties and defined as assertions in the test bench. Assertion checkers will verify the correctness of assertions by SystemVerilog. Another important feature of a test bench is the input patterns. Some test corners need special input patterns. The more input patterns in the test bench, the more, for example, lines will be covered during verification. Table 4.2 shows five test benches with different test patterns and verification times for various coverage metric reports for the RS232 benchmark with Trojan 1. Generally, the verification time will increase with more test patterns and the code

Table 4.2 Analyzing the impact of the test bench on coverage metrics (a benchmark with Trojan 1 is used)

Test bench #	Test bench 1	Test bench 2	Test bench 3	Test bench 4	Test bench 5
Test patterns #	2000	10,000	20,000	100,000	1,000,000
Verification time	1 min	6 min	11 min	56 min	10 h
Line coverage (%)	89.5	95.2	98.0	98.7	100
FSM state coverage (%)	87.5	87.5	93.75	93.75	100
FSM transition coverage (%)	86.2	89.65	93.1	96.5	100
Path coverage (%)	77.94	80.8	87.93	97.34	100
Assertion	Successful	Successful	Successful	Successful	Failure

coverage will be higher as well. For Test Bench 1 to Test Bench 4, all the coverage reports are less than 100 % and all the assertions are successful, which indicates that the Trojan is dormant during the entire verification. The special test patterns added in Test Bench 5 increase the pattern count significantly and can activate the Trojans inserted in the benchmark. Hundred percent code coverage could be achieved with these additional test patterns. If one of the assertion experiences a failure, it signifies Trojan activation and the RS232 will give an erroneous output. One can conclude that the IP is Trojan-inserted. However, it is not easy to generate a test bench with 100 % code coverage for large IPs, and the verification time will be extremely long.

This phase of the flow can help improve the quality of the test bench. Given the time-coverage trade off, Test Bench 4 is selected for further analysis.

4.6.3 Reducing the Suspicious Signals

All the 19 benchmarks with different Trojans are synthesized to generate the gate-level netlist. The removal of redundant circuits is done during the synthesis process with special constrains using the Design Compiler. The simulation results are shown in Table 4.3. The second column in the table shows the area overhead of each Trojan after generating the final layout. As the table shows, Trojans are composed of different sizes, gates, and structures, as well as different triggers and payloads as previously mentioned. The smallest Trojan has only 1.15 % area overhead. The percentage of Trojan area covered by suspicious signals *SS-Overlap-Trojan* is obtained by,

$$\text{SS-Overlap-Trojan} = \frac{N_{\text{SS}}}{N_{\text{TS}}}$$

where N_{SS} is the number of suspicious signals and N_{TS} is the number of Trojan signals. The results in Table 4.3 show that *SS-Overlap-Trojan* is between 67.7 % and 100 %, as shown in seventh column.

If all the suspicious signals are part of the Trojan, the *SS-Overlap-Trojan* would be 100 %. This indicates that the number of signals in the final suspicious list fully overlapped with those from Trojan. This is an indicator of how successful the flow is at identifying Trojan signals. In addition, if the Trojan is removed or detected by sequential ATPG, the *SS-Overlap-Trojan* would also be 100 %.

Test Bench 4 is used to verify which signals in each Trojan-inserted circuit are not covered by the simulation with all the successful assertions. Those quiet signals are identified as suspicious. The number of suspicious signals of each benchmark is shown in the third column of Table 4.3. Different benchmarks have a different number of suspicious signals based on the size of its Trojans. The larger the Trojan is, the more suspicious signals it has. On the other hand, the suspicious signals' stuck-at values are monitored by verification. All stuck-at-faults are simulated by the ATPG tool with scan chain in the netlist. If the fault is untestable, the suspicious

Table 4.3 Suspicious signal analysis

Benchmark (RS232)	Trojan area overhead (%)	Step 1: Number of SS after RCR with synthesis	Step 2: Number of SS after RCR with ATPG	Step 3: Number of SS after equivalence analysis	Step 4: Number of SS after sequential ATPG	SS-Overlap-Trojan (%)
With Trojan 1	11.18	22	20	17	12	100
With Trojan 2	20.35	17	16	3	Trojan is identified	100
With Trojan 3	10.48	20	15	15	10	97.3
With Trojan 4	20.35	3	3	3	2	87.6
With Trojan 5	4.59	9	8	8	7	100
With Trojan 6	1.15	1	1	1	Trojan is identified	100
With Trojan 7	3.79	3	3	3	2	100
With Trojan 8	1.15	1	Trojan is removed	–	–	100
With Trojan 9	3.79	3	Trojan is removed	–	–	100
TR04C13PI0	1.6	8	3	3	3	100
TR06C13PI0	1.8	9	3	3	3	100
TR0AS10PI0	2.09	8	1	1	1	100
TR0CS02PI0	25.3	59	55	39	39	67.7
TR0ES12PI0	2.09	8	1	1	1	100
TR0FS02PI0	25.0	30	28	20	20	73.3
TR2AS0API0	11.9	19	18	11	11	100
TR2ES0API0	12.0	20	18	11	11	100
TR30S0API0	12.4	22	20	13	13	93.6
TR30S0API1	12.3	25	22	14	14	87.3

circuit is a redundant circuit and will be removed from the original gate level netlist, in addition to the gates that drive the net. The number of suspicious nets after redundant circuit removal is shown in the fourth column of Table 4.3. As can be seen in the table, the suspicious nets of benchmarks with Trojan 8 and Trojan 9 are zero, which means that if the redundant circuits are removed in the two benchmarks, the benchmarks will be Trojan-free. The reason that redundant circuit removal can distinguish Trojans is that some Trojans are designed without payload and have no impact on circuit functionality. Thus it can be concluded that such Trojans can be removed by redundant circuit removal.

The remaining suspicious nets of each benchmark are needed to be processed by equivalence analysis and sequential ATPG. The fifth and sixth columns in Table 4.3 show the number of suspicious signals after the first two steps. It can be concluded that equivalence analysis can reduce a large number of suspicious signals, and sequential ATPG can be effective as well. For benchmarks with Trojan 2 and Trojan 6, the sequential ATPG can generate sequential patterns for the stuck-at faults in the suspicious signal. The sequential test patterns improve the test bench and increase its coverage percentage. Even though the coverage percentage is not 100 %, some assertions experience failure during simulation. Thus, the benchmarks with Trojan 2 and Trojan 6 are identified as Trojan-inserted.

The flow is implemented on ten trust benchmarks from the Trust-Hub (http://trust-hub.org/resources/benchmarks) and the results reported in rows 11–20 in Table 4.3 show that the presented flow can effectively reduce the total number of suspicious signals. In addition, as shown in seventh column, there is a good overlap between the number of suspicious signals and the actual Trojan signals inserted into each benchmark. However, some benchmarks experience low *SS-Overlap-Trojan*, such as RS232-TR0CS02PI0, since only part of this Trojan was activated during simulation.

4.6.4 Trojan Coverage Analysis

In the suspicious list, not all of signals are a result of Trojans. However, the *TriggerEnable* signal must be in the suspicious list if the IP contains a Trojan. Once one net is identified as part of a Trojan, it can be concluded that the 3PIP is Trojan-inserted. All the gates driving this net are considered to be Trojan gates. Figure 4.8 shows that the percentage of Trojan signals in the suspicious list increases significantly with the flow. As the authors apply different steps (step 1–4) to the benchmarks, 72 %, on average, of the suspicious signals are of the result of Trojans after redundant circuit removal with synthesis and ATPG in the 19 benchmarks. However, the percentage increases to 85.2 % when equivalence analysis is done and 93.6 % of signals in the suspicious signal list come from Trojans after sequential ATPG is applied to these benchmarks.

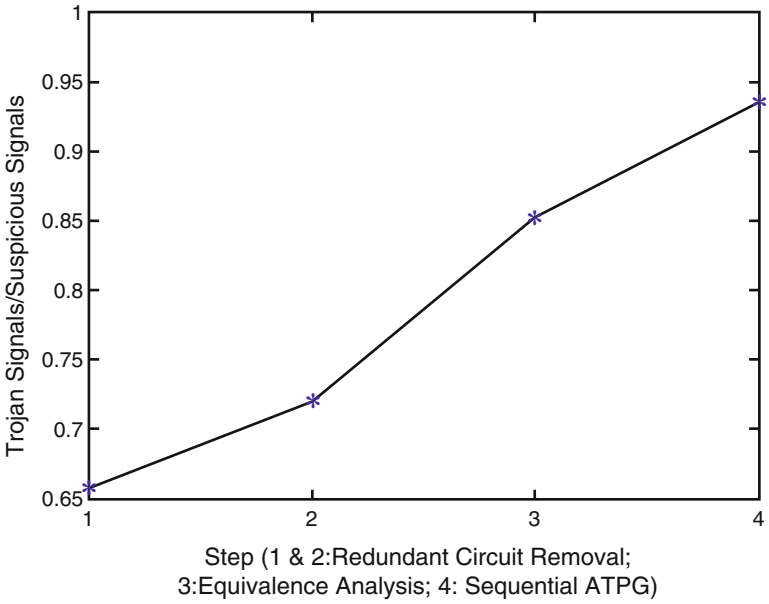


Fig. 4.8 Average Trojan signals/suspicious signals in 19 benchmarks

4.7 Conclusion

In this chapter, we have presented a technique to verify the trustworthiness of 3PIPs. This technique involves formal verification, coverage analysis, redundant circuit removal, sequential ATPG, and equivalence theorems. The code coverage generates the suspicious signals list. Redundant circuit is removed to reduce the number of suspicious signals. Equivalence theorems are developed for the same purpose. Sequential ATPG is used to activate these suspicious signals and some Trojans will be detected. However, more work is needed to get 100 % hardware Trojan detection rates in 3PIPs.

References

1. Report of the Defense Science Board Task Force on High Performance Microchip Supply, Defense Science Board, US DoD (2005), http://www.acq.osd.mil/dsb/reports/2005-02-HPMSi_Report_Final.pdf
2. M. Tehranipoor, F. Koushanfar, A survey of hardware Trojan taxonomy and detection. IEEE Des. Test Comput. **27**(1), 10–25 (2010)
3. M. Tehranipoor, C. Wang, *Introduction to Hardware Security and Trust* (Springer, New York, 2011)

4. H. Salmani, X. Zhang, M. Tehranipoor, *Integrated Circuit Authentication: Hardware Trojans and Counterfeit Detection* (Springer, Cham, 2013)
5. X. Zhang, M. Tehranipoor, Case study: detecting hardware Trojans in third-party digital IP cores, in *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2011)
6. VSI Alliance, VSI Alliance Architecture Document: Version 1.0 (1997)
7. DIGITIMES. Trends in the global IC design service market (2012). Retrieved from http://www.digitimes.com/news/a20120313RS400.html?chid=2
8. M. Tehranipoor, et al., Trustworthy hardware: Trojan detection and design-for-trust challenges. Computer **44**(7), 66–74 (2011)
9. K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, M. Tehranipoor, Hardware Trojans: lessons learned after one decade of research. ACM Trans. Des. Autom. Electron. Syst. **22**(1), Article 6 (2016)
10. M. Bilzor, T. Huffmire, C. Irvine, T. Levin, Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage, in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2012)
11. C. Sturton, M. Hicks, D. Wagner, S. King, Defeating UCI: building stealthy and malicious hardware, in *2011 IEEE Symposium on Security and Privacy (SP)* (2011), pp. 64–77
12. C. Cadar, D. Dunbar, D.R. Engler, Klee: unassisted and automatic generation of high-coverage tests for complex systems programs, in *Proceedings of the 2008 USENIX Symposium on Operating Systems Design and Implementation* (2008)
13. A. Biere, A. Cimatti, E. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in *Proceedings of the ACM/IEEE Annual Design Automation Conference* (1999), pp. 317–320
14. A.C. Myers, B. Liskov, A decentralized model for information flow control, in *Proceedings of the 1997 Symposium on Operating Systems Principles* (1997)
15. E. Love, Y. Jin, Y. Makris, Proof-carrying hardware intellectual property: a pathway to trusted module acquisition. IEEE Trans. Inf. Forensics Secur. **7**(1), 25–40 (2012)
16. Y. Jin, B. Yang, Y. Makris, Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing, in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2013)
17. G. Xiaolong, R.G. Dutta, Y. Jin, F. Farahmandi, P. Mishra, Pre-silicon security verification and validation: a formal perspective, in *Proceedings of the 52nd Annual Design Automation Conference* (ACM, New York, 2015), p. 145
18. S. Bhunia, M.S. Hsiao, M. Banga, S. Narasimhan, Hardware Trojan attacks: threat analysis and countermeasures. Proc. IEEE **102**(8), 1229–1247 (2014)
19. J. Rajendran, V. Vedula, R. Karri, Detecting malicious modifications of data in third-party intellectual property cores, in *Design Automation Conference (DAC)* (2015)
20. J. Rajendran, A.M. Dhandayuthapany, V. Vedula, R. Karri, Formal security verification of third party intellectual property cores for information leakage, in *29th International Conference on VLSI Design* (2016)
21. H. Salmani, M. Tehranipoor, Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level, in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (2013), pp. 190–195
22. H. Salmani, R. Karri, M. Tehranipoor, On design vulnerability analysis and trust benchmarks development, in *Proceedings of IEEE 31st International Conference on Computer Design (ICCD)* (2013), pp. 471–474
23. M. Tehranipoor, H. Salmani, X. Zhang, *Integrated Circuit Authentication: Hardware Trojans and Counterfeit Detection* (Springer, Cham, 2013)
24. S. Bhunia, M.S. Hsiao, M. Banga, S. Narasimhan, Hardware Trojan attacks: threat analysis and countermeasures. Proc. IEEE **102**(8), 1229–1247 (2014)
25. R.S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, S. Bhunia, MERO: a statistical approach for hardware Trojan detection, in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'09)* (2009)

26. A. Waksman, M. Suozzo, S. Sethumadhavan, FANCI: identification of stealthy malicious logic using Boolean functional analysis, in *Proceedings of the ACM Conference on Computer and Communications Security* (2013), pp. 697–708
27. J. Zhang, F. Yuan, L. Wei, Z. Sun, Q. Xu, VeriTrust: verification for hardware trust, in *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference* (2013), pp. 1–8
28. J. Zhang, F. Yuan, Q. Xu, DeTrust: defeating hardware trust verification with stealthy implicitly-triggered hardware Trojans, in *Proceedings of the ACM Conference on Computer and Communications Security* (2014), pp. 153–166
29. J. Rajendran, O. Sinanoglu, R. Karri, Building trustworthy systems using untrusted components: a high-level synthesis approach. IEEE Trans. Very Large Scale Integr. VLSI Syst. **24**(9), 2946–2959 (2016)
30. Synopsys, The Synopsys Verification Avenue Technical Bulletin, vol. 4, issue 4 (2004)
31. I. Ugarte, P. Sanchez, Formal meaning of coverage metrics in simulation-based hardware design verification, in *IEEE International High Level Design Validation and Test Workshop (HLDVT)* (IEEE, Napa Valley, 2005)
32. M. Bushnell, V. Vishwani, *Essentials of Electronic Testing for Digital, Memory and Mixed Signal VLSI Circuits*, vol. 17 (Springer Science & Business Media, 2000)

Chapter 5
Analyzing Circuit Layout to Probing Attack

Qihang Shi, Domenic Forte, and Mark M. Tehranipoor

5.1 Introduction

Physical attacks have caused growing concern for design of integrated circuits used in security critical applications. Physical attacks circumvent encryption by attacking their silicon implementations. IC probing is one form of physical attack that allows an attacker to access security critical information by directly accessing physical wires in the IC that carry such information [1]. For convenience, we henceforth refer to such physical wires that probing attacks target as *targeted wires*. Successful probing attacks have been reported on smartcards and microcontrollers in mobile devices [2, 3]. In a successful probing attack, plaintexts such as personal data, software-form IP, or even encryption keys can be compromised [4].

IC probing attacks happen for a spectrum of different reasons in the wild. Probing attacks circumvent encryption without rendering the targeted device inoperable, and therefore they enable unauthorized access on systems carrying sensitive information. An otherwise innocuous tech-savvy kid might want to hack his satellite TV smartcard to "unlock" channels he isn't supposed to watch [5]; The issue becomes less innocuous when authorities seize multi-million dollars' worth of asset in connection with piracy lawsuits [6]. As modern lives become more integrated with the Internet through mobile hardware, security of private data, and keys stored on these devices also raised concern. If leaked, such information can be used to further identity theft, blackmail, or a number of other threats to individual rights and liberties. The worst-case scenario is probably for systems that enable access

Q. Shi (✉)
ECE Department, University of Connecticut, Storrs, CT, USA
e-mail: qihang.shi@engr.uconn.edu

D. Forte • M.M. Tehranipoor
ECE Department, University of Florida, Gainesville, FL, USA
e-mail: dforte@ece.ufl.edu; tehranipoor@ece.ufl.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_5

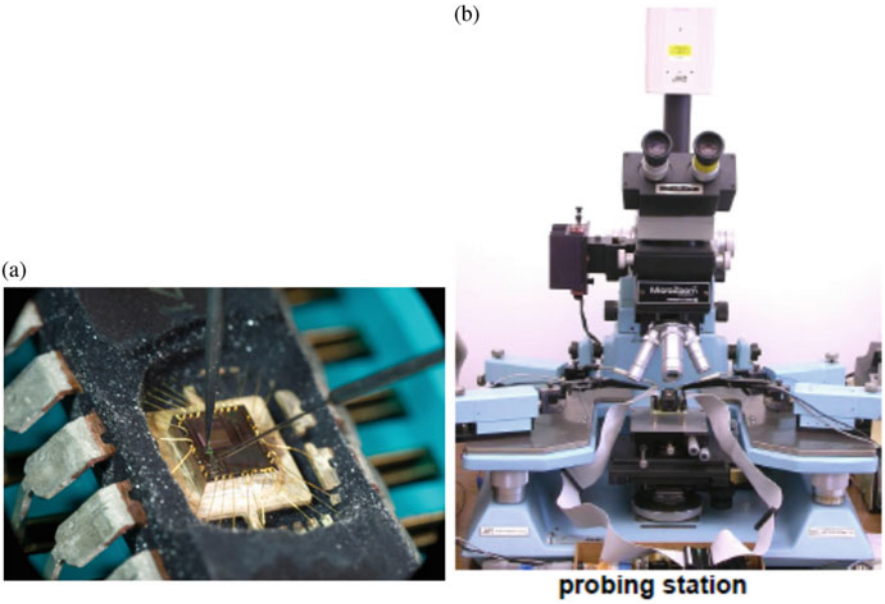


Fig. 5.1 Example of a device under probing and probing station. (**a**) An IC under probing, with package partially removed and probing pin inserted [7]. (**b**) A probing station [1]

to information related to public security—for example, the security tokens used in defense industries—to become compromised. Therefore, it follows that tamper resistance has become a default requirement in US defense acquisitions [8].

A typical probing attack involves (at least partially) remove package of the targeted device (Fig. 5.1a), place it on a probing station (Fig. 5.1), and position the metal probe to form an electrical contact with targeted wires (also shown in Fig. 5.1a). In real attacks, targeted wires are usually found by reverse engineering a sacrificial device of the same design. Reverse engineering can be a lengthy process, however, it can be accelerated if the IC under attack reuses compromised hardware IP blocks, i.e., hardware IP blocks that have been reverse engineered in previous attacks. In most IC designs, targeted wires are often buried beneath layers of passivation, dielectric and other metal layers. As an example, consider a scenario where an attacker targets a wire routed no higher than Metal 4 layer, in an IC that has the same cross-section as in Fig. 5.2. To initiate a probing attack, an attacker has to expose targeted wires either by milling through all layers above them, or the silicon substrate beneath them. In our example, he will have to mill either from passivation layer down to Metal 4, or from substrate up to Metal 4. In either scenario, use of milling tools is necessary.

Most security critical ICs are reinforced against probing attacks with active shield. This approach functions by detecting milling events and sending alarms once a breach has been detected. Here "sending alarm" is a general term referring

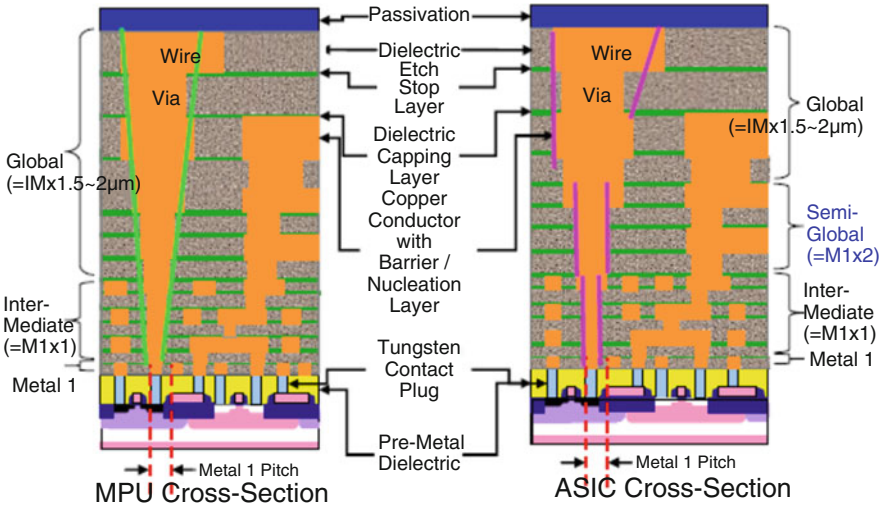


Fig. 5.2 Typical cross-sections of microprocessors (MPU) and Application-Specific Integrated Circuits (ASIC) [9]

to all proper security actions to address a probing attempt, such as zeroizing all sensitive information. Due to their popularity in research and practice, attacks and antiprobing designers often focus their efforts on discovering and improving exploits and countermeasures to penetrate or prevent penetration of the active shield.

In this chapter, we first review technologies and techniques known for their use in probing attacks. These include technologies developed for other purposes such as IC failure analysis, and specifically probing techniques such as back-side attacks. These will be covered in Sect. 5.2. Based on this knowledge, we then investigate approaches to secure designs against probing attacks, commonly known as *antiprobing* designs. This is covered in Sect. 5.3, which introduces published proposals to secure the design against probing attacks, their known problems and research to address these problems. Based on background knowledge provided in these two sections, it becomes apparent to us that evaluation of designs in terms of their vulnerabilities to probing attacks would likely prove a valuable contribution to the field. Therefore, in Sect. 5.4 we present a layout-driven framework to assess designs' vulnerabilities to probing attacks, rules of said assessment framework, a set of assumptions on state-of-the-art antiprobing designs, and an algorithm to quantitatively evaluate design for exposure to probing attacks. Finally, the chapter is concluded in Sect. 5.5.

5.2 Microprobing Attack Techniques

In this section we introduce various techniques used in probing attacks. All probing attacks will involve milling to expose targets for the probe to access; however, it is also important to acknowledge that a probing attack is a concerted effort that involves different techniques at various stages, and many techniques other than milling help shape the threat and limitations of the probing attack. Understanding of these techniques is essential to the understanding of principles when designing against such attacks.

5.2.1 Essential Steps in a Probing Attack

Before introducing specific techniques for probing attacks, let's first take a look at the *modus operandi* of a typical probing attack. In surveying reported attacks, we have found probing attacks require at least four essential steps [3], each must be successful for the attack to succeed (shown in Fig. 5.3 as different rows):

- Reverse engineer a sacrificial device to get its layout and find target wires to microprobe;
- Locate the target wires with milling tool;
- Reach the target wires without damaging target information;
- Extract target information.

Each step can have a number of alternative techniques where success with only one of them is necessary. For example, locating target wires in layout can be done by reverse engineering the design or with information from a similar design. Obfuscation can force the attacker to spend more time on this step, for example, by performing dynamic address mapping and block relocation on embedded memories [10] or by performing netlist obfuscation using modification kernel function [11]; but if the same hard IP is reused in another design both designs become vulnerable once that IP becomes compromised. The diagram here lists all the alternatives we know to exist, and new attacks can be easily integrated.

Shown in Fig. 5.3 is a typical flow of a probing attack, where each step is shown in a row and each block shows an alternative technique to complete that step. Some techniques are shaded with patterns to represent the particular capability to enable that technique. *Disable shield* technique is shown with two blocks each having two patterns to show it can be completed either with circuit editing or fault injection, but in both options reverse engineering is required. Techniques in white boxes that do not have a patterned alternative show possible exploits from avoidable design flaw rather than lack of protection. For example, "Use shield to help fine navigation" is possible if shield wires were not placed in 45° with regard to functional routing [3]; and if no internal clock source is used, attacker could simply "stop external clock" to extract all information without having to use multiple probes.

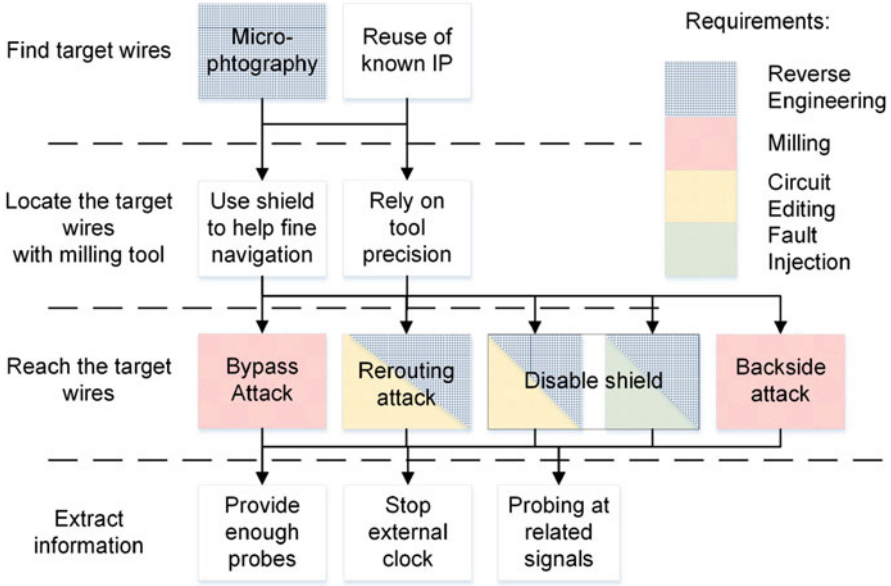


Fig. 5.3 Diagram of known probing techniques for assessment of design vulnerability

5.2.2 Microprobing Through Milling

On ICs fabricated with feature dimensions larger than 0.35 μm, laser cutters can be used to remove these layers [1]. For technologies of lower dimensions, currently the state-of-the-art tool is a technology called the Focused Ion Beam (FIB) [12]. With the help of FIB, an attacker can mill with sub-micron or even nanometer level precision [13]. Aspect ratio (see Fig. 5.4b) is a measure of the FIB performance defined as the ratio between depth D and diameter d of the milled hole [14]. FIB instruments with higher aspect ratio can be expected to mill a hole of smaller diameter on the top-most exposed layer, and therefore leave smaller impact on all the other circuitry the attacker is not interested in breaking. When milling in nanometer scale and applied on silicon ICs, state-of-the-art FIB systems can reach an aspect ratio up to 8.3 [15]. In addition to milling, FIB is also capable of depositing conducting traces [16], which adds Circuit Editing (CE) to the attacker's repertoire.

The most straightforward way to expose targeted wires through milling is to mill from the Back End of Line (BEOL), i.e., from passivation layer and top metal layer towards silicon substrate (see Fig. 5.4a). This is called a *front-side attack*. An obvious disadvantage of front-side attack is that targeted wires may be covered by other wires above it, and without thorough reverse engineering of the IC the attacker cannot know for sure whether cutting through these covering wires would corrupt the information he seeks to access. In real attacks, attackers address this problem by focusing on bus wires and try to probe from more exposed (i.e., not covered by

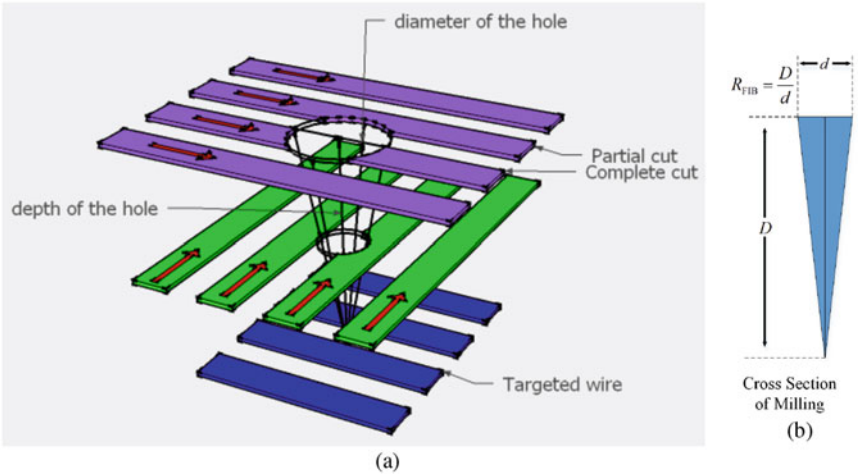


Fig. 5.4 Milling with Focused Ion Beam (FIB) technology. (**a**) Milling through covering wires to expose targeted wires. (**b**) Definition of aspect ratio of an FIB

other wires on higher layers) segments of the targeted wires [2]. The latter could be facilitated by performing reverse engineering a sacrificial device, in which case desirable milling sites can be found from exposed layout of the sacrificial device. The attacker still needs to find a way to navigate to the same sites on the target device, which might be simplified or complicated depending on whether special care was taken against this step by the designer.

5.2.3 Back-Side Techniques

In addition to milling through routing layers and inserting metal probes at targeted wires, other techniques exist. Another way is through the silicon substrate, the so-called *back-side* attacks [17]. In addition to probing at wires, back-side attacks can also access at current activities in transistor channels (bottom layer in Fig. 5.2). This is facilitated by techniques known as Photon Emission (PE) and Laser-based Electro-Optical Modulation (EOFM or Laser Voltage Techniques LVX) [18]. Between the two methods, PE can be passively observed without any external stimulation, while LVX requires IR laser illumination of the active device. On the other hand, PE manifests mainly during rise and fall time of the signal, while LVX response is linearly correlated to the voltage applied to the device. Both methods are very reliable as modern IC debug and diagnosis tools, and their legitimate uses ensure that the available tools will maintain pace with semiconductor scaling. Indeed, recent reports show that all 16 bytes encryption keys of AES-128 can be recovered in 2 h [19].

One limiting factor of the passive techniques is that both methods require observation of photon emissions, which makes them limited by the wavelength of emitted photons. For example, detection of PE on majority of instruments operate between 2 and 4 μm [20]. Depending on spatial distribution of switching activity, this might cause a problem on devices manufactured with deep sub-micron technologies. As technology node advances and feature size shrinks, emissions, especially responses from LVX techniques from more devices will become indistinguishable, thus making probing attacks from back-side difficult [18].

Another possibility of back-side attacks is circuit editing, enabled with FIB. Like with front side, wires can be cut from back-side; and in addition to cutting wires, it is also possible to deposit conductive material in holes dug into drain and source regions of transistors, serving as metal probes that can be inserted anywhere without worrying cutting open any covering wires as the front-side attack [21].

Back-side attacks are harder to defend against since conventional IC design process doesn't place anything beneath the silicon substrate. However, next generation security critical designs may choose to fabricate a *back-to-back* 3D IC to avoid leaving the silicon substrate exposed [22, 23], effectively eliminating back-side attacks all together. Therefore, protection against front-side attacks remains an important topic for antiprobing designs.

5.2.4 Other Related Techniques

This category summarizes all techniques that may be used to help further the probing attack or jeopardize a secured IC's defense against it. Some techniques can be quite essential to the task: for example, all probing attacks have to consist of some amount of reverse engineering. At least, the attacker has to locate targeted wires (or in the case of back-side attacks, targeted gates) by discovering which part of the circuit is responsible for carrying the sensitive information the attacker seeks; if cutting open a functional wire is inevitable, additional reverse engineering is required to find out cutting open which wires do not lead to corruption of the information the attacker wishes to probe. The problem with reverse engineering is the amount of work and time cost involved: since the target of probing attacks are often information that has a short life span (e.g., keys and passwords), this can sometimes be used to work against the attacker.

Some other techniques are supplemental to the main attack. One example is circuit editing. An obvious use of this technique can be used to disable the security feature, e.g., cutting off the alarm bit and rewiring it to power supply or ground. In addition to that direct approach, circuit editing could also be used to supplement other requirements, for example, clock source can be rewired to a controlled source, which can be very useful when the number of wires is too many for available probes.

5.3 Protection Against Probing Attacks

This section gives an overview of current research in securing designs against probing attacks. The focus of this section is on active shields, a protection mechanism that has received the central attention of the academia, and application on security hardware in the market. Much research has been aimed at finding shield exploits and providing fixes to stop known exploits. In addition to attack and defense with regard to active shields, we also present a mathematical model of active shield and its ability to detect milling. Other approaches that are not active shields and design details that complement active shields are also covered.

5.3.1 Active Shields

The most common method to protect IC from probing attack is the active shield, which detects milling by placing signal-carrying wires on top metal layers [22, 24–28]. The design is called an active shield because the wires on top metal layers are constantly monitored in order to detect an attack.

Figure 5.5 is one example constructed for the sake of illustration. The detection system generates a digital pattern with a pattern generator, sends them through mesh wires on top metal layer, and then compares received signals with copies from lower

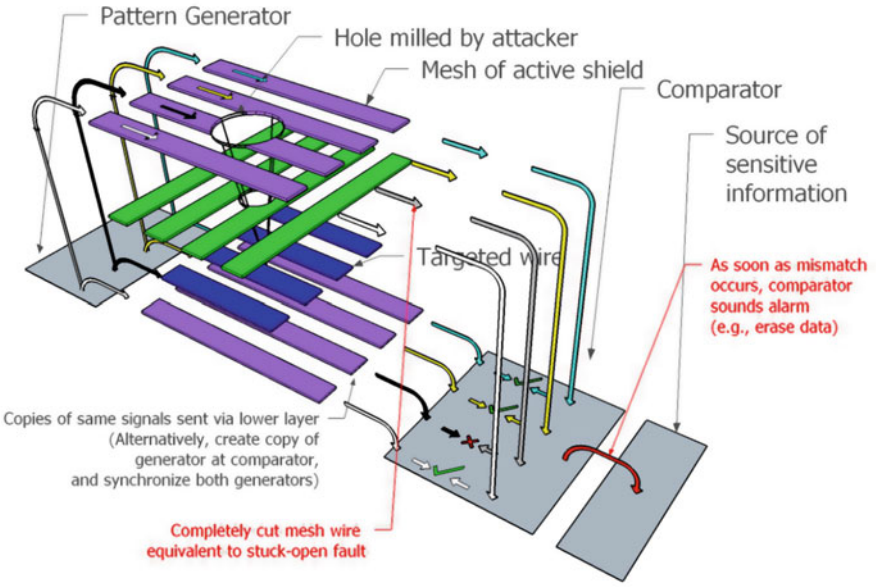


Fig. 5.5 Example of active shield for the purpose of milling detection

layer. If an attacker mills through the top layer mesh wires to reach targeted wire he will cut open some of them and cause a mismatch at the comparator. This will trigger the alarm, for example, stopping the generation of or destroying sensitive information.

Active shield as shown in Fig. 5.5 is known as a *digital* active shield, since it uses comparisons between digital signals to detect milling activity. *Analog* active shields also exist: for example, the authors in [24] use capacitance measurement on top layer shield mesh wires to detect damage done to it, and thereby detect tampering. A similar design [29] utilizes conductivity change in dielectric material due to exposure to ion irradiation, an essential step in FIB milling. Another design [25] uses a delay chain as a reference to compare against shield wire RC delay, and issues an alarm when mismatches between them are discovered. The problem with analog shield designs is that analog sensors rely on parametric measurement, which has been shown to become more difficult due to a number of issues, such as elevated process variation with feature scaling, improved milling capabilities that cause less disturbance, etc. [3]. These issues contribute to a higher error rate and make detection unreliable, thereby making analog active shield a less preferable approach.

5.3.2 Techniques to Attack and Secure Active Shields

Despite its popularity, active shields are not without problems. There are two main weaknesses of this approach, namely:

1. Active shields require large routing overhead, usually at least one entire layer;
2. Active shields have to be placed on top metal layers, which might not always be the most suitable layer.

We provide detailed discussions on both issues in the following subsections.

5.3.2.1 Routing Overhead

For thorough protection, active shields have to completely occupy at least one metal routing layer, which can be quite costly to implement. This is thought to be necessary because as long as attackers can afford extra probes, it is possible for him to reconstruct the desired signal from other related signals if they are not protected as well [30]. This requirement does not go well with designs having tight cost margin, or designs with few routing layers, which is especially true for devices such as smartcard, which are often fabricated with technology of larger dimensions such as 350 or 600 nm [2]. In fact, ICs designed for security critical applications such as smartcards, microcontrollers in mobile devices, and security tokens [2, 3, 31] are among the most common victims to this kind of attack. Lacking a very wide cost margin or a lot of routing layers precludes these devices from a number of new shield

designs that often incur area or routing overhead. Meanwhile, these same devices are in greatest need of evaluation to have realistic protection standards. Indeed, all designs eventually become outdated as new attacks are discovered, and these legacy generations of devices need up-to-date evaluation as well.

5.3.2.2 Stuck on Top Metal Layer

The "entire layer" requirement also leads to another problem, which is active shields must be placed on top-most metal layers. Due to the "entire layer" requirement, functional routings cannot penetrate shield layer without leaving an opening on the shield, therefore all routing layers above a shield layer will be unavailable to the functional design as well. Placing the shield on top-most metal layers saves routing layers for the functional design, but might come at cost of shield performance. Since top layers might not be best layers for them. In fact, top routing layers are known to have much larger minimum wire widths [32], making them less protective than lower layers [33].

Due to the popularity of the active shield approach, techniques have also been developed by attackers to neutralize it. One particularly expedient attack is called the *bypass attack* (Fig. 5.6). This attack utilizes an FIB milling tool with high aspect ratio, which allows the attacker to mill a hole so thin it's not going to completely cut off any mesh wires, therefore evading detection. Bypass attack is often the preferred method by the attacker because it does not require additional reverse engineering or circuit editing to reach the targeted wires, which saves his time and cost. Because of its advantage in speed, bypass attack is an especially serious challenge that all active shields must properly address. Another method exploits the fact that active shield functions by checking signals as received at end of shield wires. If the correct signals can be replicated, then the shield could be fooled. This is called a *reroute attack*. There can be a couple of ways to do this. One simpler approach is made possible when routing pattern of the shield wires allows rerouting, i.e., creating a short-cut with FIB so that a section of shield wire is made redundant [2], as shown in Fig. 5.7. That section of shield wire can then be removed without impacting signals received by the active shield, effectively creating an opening for probing. Another possible scenario is that attacker might reverse engineer the pattern generator so that correct signals can be fed from off-chip. In this case, the whole wire becomes redundant.

New active shield designs have been proposed to prevent these exploits. The authors in [22] investigated the problem of an attacker predicting the correct signals if the random vector generation is not secure enough. The authors then presented a design where block ciphers in Cipher Block Chaining (CBC) mode are used to generate secure random vectors, whose seed comes from memory and fed by software. Another research proposed to obfuscate layout routing of the active shield so that the attacker would not be able to figure out how to perform a successful reroute attack [27].

Perhaps the most serious threat to active shields comes from the capability of FIB technology itself. The capability of depositing metal as well as removing them

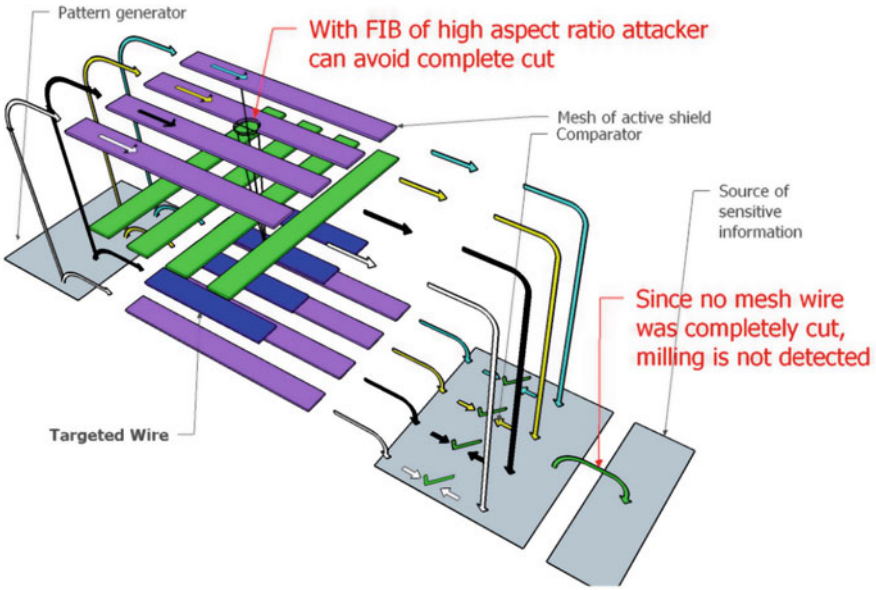


Fig. 5.6 Example of a bypass attack on active shield

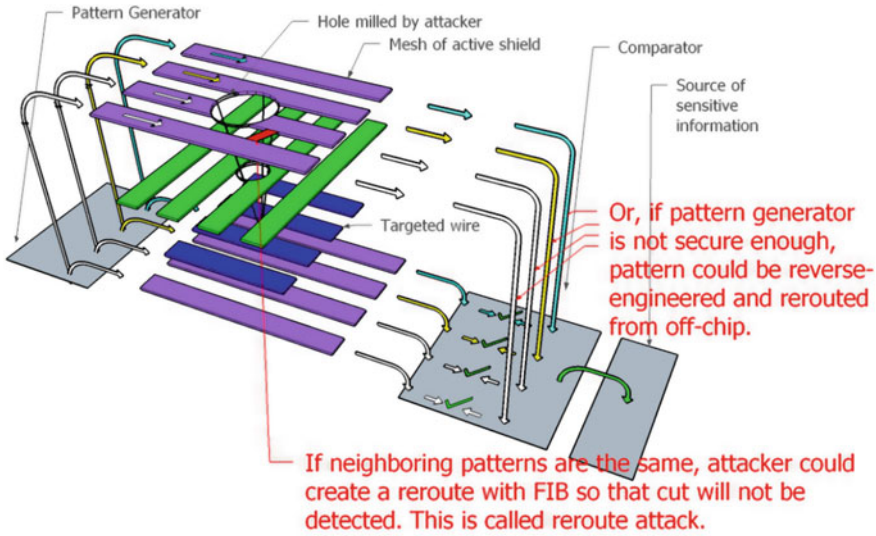


Fig. 5.7 Example of a reroute attack on active shield

effectively allows circuit editing, which enables the attacker to directly disable the active shield by editing its control circuitry or payload if it proves too difficult to bypass [3].

5.3.3 Other Antiprobing Designs

In addition to active shield designs, other approaches also exist. Authors in [34] presented a design to detect acts of probing by monitoring change of capacitance on security critical nets, as a cheaper alternative to the more popular active shield method as it requires far less area and routing overhead. However, it can only protect a certain number of specified nets from probing attacks using metal probes. Additionally, protection on certain specified nets could only be circumvented by probing at related signals instead [30]. Cryptographical techniques have also been proposed to address the probing attack. One cryptographical method called *t*-private circuits [35] proposed to modify the security critical circuit so that at least $t + 1$ probes are required by an attacker to extract one bit of information. This method is cryptographically sound and does not require detection to deter probing, which eliminates the problem of protection design itself being disabled by the attacker. The proposed solution is also helpful in defeating side-channel attacks (SCA) and inspired further research in that area [36]. The weakness of this approach is it does incur quite large area overhead, and reliance on secure random number generation opens up questions on what happens when the random number generator is compromised, for example, with a probing attack. It has also been shown that such an approach might be jeopardized during CAD optimization process [37].

Some other approaches are often seen in security products in the market but less mentioned in academic literature. These approaches receive more discussion from researchers who publish successful attacks against security hardware in the market. For example, many security devices in the field also perform encryption of memories, for which a common problem is insufficient security when storing the keys for the encryption [31]. Some use one-time programmable (OTP) memories which can be read optically, or Electrically Erasable Programmable Read-Only Memories (EEPROM) which can be reprogrammed with ultra-violet (UV) light [38]. Other examples include scrambling of bus wires in the layout as a way to throw off attackers, password protection for boot strap loaders, and using one single sense signal for serpentine wires that elbow and bend to cover the layout [31]. These designs might not be as effective as they intended—since our knowledge of them is from their failure—but could serve as examples of "Do Nots" in antiprobing designs.

Another category consists of single-issue remedies that seek to disrupt specific steps during probing attacks. Many such techniques exist as recommendations by researchers who publish successful attacks [2, 3, 31]. These recommendations include methods to disrupt positioning of FIB for milling, avoid reusing hardware IPs to prevent attackers from copying existing reverse engineering knowledge against compromised IPs, methods to make IC packages more resistant to decapsulation, etc. Although not solutions on their own, these recommendations provide important insights into principles from the other side of the problem.

Table 5.1 Performance against known probing techniques of published designs

Designs	Protection against					
	Bypass shield	Rerouting attack	Disable shield	Back-side attack	Prediction attack	Related signals
Analog shield	Weak [3]	No	No		N/A	Yes
Random active shield [27]	Yes	Yes	No		No	Yes
Cryptographically secure shield [22]	Yes	Yes	No		Yes	Yes
PAD [34]	N/A	N/A	No	Some[a]	N/A	No

[a]PAD works for back-side attacks that require electrical contact to the targeted wires. It does not prevent passive back-side attacks such as PE

5.3.4 Summary on Antiprobing Protections

Based on known probing techniques (as was shown in Fig. 5.3) we may assess the protection of a few published designs, as shown in Table 5.1.

5.4 Layout-Based Evaluation Framework

In this section, we provide a motivation to investigate methods to evaluate antiprobing designs. We then discuss principles of antiprobing designs and rules to assess them, based on reviewing *modus operandi* of probing attacks. A layout-driven algorithm based on mainstream layout-editors is then introduced, which provides a quantitative evaluation of designs' vulnerabilities to probing attacks by reviewing their layout.

5.4.1 Motivation

Existing research on active shield designs focuses on preventing exploits. However, major problems exist with this approach. It negates the need of mass-produced and legacy generation security products of having realistic protection evaluations, and gives us a false sense of security despite the fact that security is always relative. Should designers focus on beating latest exploit into current shield designs, inherent problems like these tend to get overlooked. Further, circuit editing capability of the FIB actually makes it impossible for active shields to have absolute security: attacker can always choose to edit out the shield itself. Despite that, having a design

that beats all known exploits can make people dangerously comfortable with a false sense of security and forget that fact. This reality has put the ability to evaluate a design for vulnerability to probing attacks in dire need.

Evaluation contributes to existing antiprobing design flow in a few ways (Fig. 5.8):

1. First, the evaluation tool can be used to create a feedback for the design process, and help designers in pursuit of an optimized design (Fig. 5.8a).
2. In addition, comparisons can be made between security evaluations of certain representative designs. By carefully controlling design methods and parameters, design principles such as trade-offs in antiprobing designs can be identified and investigated (Fig. 5.8b).
3. In addition to identifying design principles, evaluation also enables us to study new design ideas, and optimize them into new design strategies, or establish a solid understanding of why they wouldn't work (Fig. 5.8c).

5.4.2 Assessment Rules

Before presenting the framework to assess protection designs against probing attacks, it is essential to establish the principles of antiprobing designs. Otherwise, research could become sidetracked by objectives unnecessary or insufficient, and assessment would lack a standard for comparison.

One pitfall for the designer might be to underestimate the capability of the attacker. When considering tools available to a probing attack, it is important to remember that attackers capable of nanometer scale milling are not restricted to probing alone. FIB itself allows circuit editing, which enables attacker to disable the whole shield by tying its detection bit to ground. Lasers can be used to inject arbitrary values to confuse protective mechanism. Indeed, both techniques have been reported successful [3]. As a result, while designs that can defeat all known attacks might not be impossible, it is certainly impractical to pursue for most devices.

Meanwhile, another myth is to underestimate the difficulty of probing attack. It is important to remember that attackers are likely to find a way in does not mean protection design is futile. The goal of a probing attack is sensitive information, and sensitivity decays with time. Information expires. Passwords are rotated. Backdoors are fixed with security updates. Even functional designs are phased out of market by new generations. Therefore, if delayed long enough, objectives of even an attacker with infinite resources can be denied.

In addition to delaying the most well-equipped attackers, it is also in the interest of the designer to deter less well-equipped attackers. This is especially true for low-cost devices such as security tokens and smartcards. This deterrence can be performed in terms of capability or information. Countermeasures vulnerable to

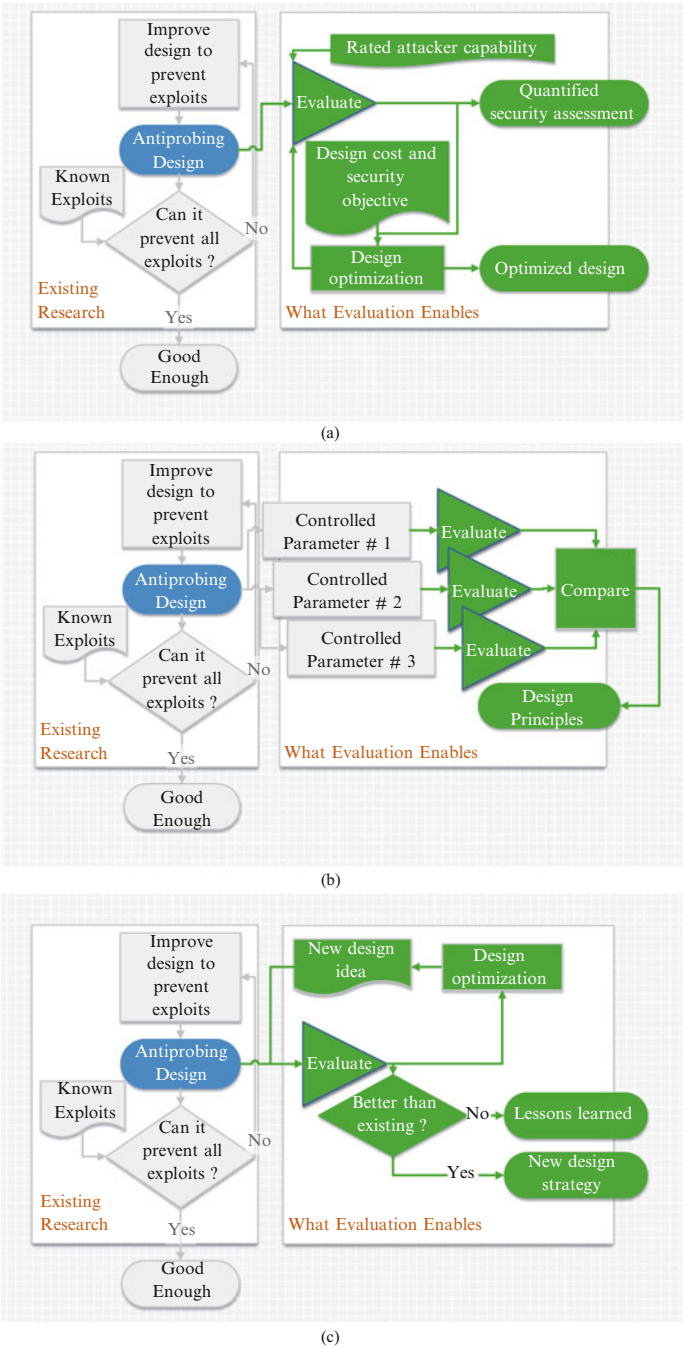


Fig. 5.8 Applications of evaluation in antiprobing design

the most cutting edge instruments might still filter out attackers that do not have access to such capabilities, and using custom designs instead of IPs reduce the risk of having a vulnerability when an IP you use is successfully attacked.

Based on principles above we propose the following rules to assess a design for vulnerability to probing attacks:

- For each necessary step during a probing attack, enumerate all known alternative techniques, the capability required by this technique, whether and how much does the design change the expected time cost on each technique;
- Represent the protection against attackers with infinite resources by the sum of techniques with the lowest time cost from each necessary step;
- For protection against less well-equipped attackers, repeat the same process without techniques requiring unavailable capabilities.

Under these rules it is possible for a particular probing technique to have an infinite time cost against a particular design: for example, an active shield with wires too thin for current FIB to bypass. However, infinite time cost is unlikely to appear for an entire step due to existence of very powerful techniques such as circuit editing: in the aforementioned example, the attacker could opt to remove the shield and disable it by fault injection or circuit editing at shield control or payload circuitry, a technique known as *disabling shield* [3].

From the assessments of existing antiprobing designs we can see that layout is of central importance in both restricting the attacker's options and increasing his time cost. If area exposed to milling can be conveniently found, it will enable designers to create antiprobing designs with better all-around resilience. For this purpose, we present an algorithm to evaluate and find *exposed area* of targeted wires.

5.4.3 State-of-the-Art Active Shield Model

To accurately evaluate exposed area of targeted wires in a layout, a mathematical model of detection mechanism is necessary. And to choose a proper mathematical model, one must first establish a set of reasonable assumptions about the detection system. The complete mathematical equations have been covered in [33]; in this section we place the emphasis on establishing a set of assumptions that define a reasonably well-designed active shield to the best of our knowledge based on published research in the area.

In this study, we assume attacker performs front-side milling with FIB technology as shown in Fig. 5.4a. The hollowed-out cone shown in the figure represents a hole milled with FIB equipment. In reality, a milled hole for the purpose of probing attack will probably be larger than the hollowed-out cone, since the probes need to maintain a reliable connection. Here we consider the cone shown in Fig. 5.4a as a best-case scenario for the attacker and worst-case scenario for the designer, so that a margin of safety can be left in favor of the shield.

Active shield designers are interested in the scenario where the attacker would make a mistake and completely cut off (at least) one shield wire. This *complete cut* event is desirable because it will make detection easy and robust. It is possible that a *partially cut* event may be detected in ways similar to the analog shield idea [24]; however, designers usually only consider complete cuts, and with good reason.

From an electrical engineering point of view, a partially cut wire creates a point of high resistivity and current density on the wire. For the timescale relevant to the probing attack and its detection, this manifests mainly as increased delay on the timing arc of the shield wire. On-chip timing measurement can be done with a frequency counter, a vernier delay chain, a current ramp and an analog-to-digital (A/D) converter, or a time-to-voltage converter [39]. None of them is known for being low on area overhead, or even close to the area overhead of a simple XOR gate per wire. Further, consider the requirement for the active shield to cover the entire functional layout, or at least all wires that might leak sensitive information if probed, the area overhead problem is astronomically worsened. To keep the area overhead manageable, such a measurement have to be handled either by a shared measurement module switched among all shield wires, or by only monitoring a few targeted wires, as is done in Probe Attempt Detector (PAD) [34]. The disadvantage of the PAD approach is discussed in Sect. 5.3.3; For the switched solution, each wire has to be compared against its own normal state and incur astronomical area overhead on memories, or use a constant reference and risk false alarms and/or escapes due to environmental and fabrication variation. Margin will have to be left to accommodate false alarms, then the same margin could also be used by an attacker to slip in undetected. In either case, much is traded for possible improvement of slightly higher FIB aspect ratio the shield is probably able to detect, depending on environmental variations.

In summary, partial cut detection greatly complicates the problem for dubious gains. The few (one, in fact) attempts that tries to do this to our knowledge [25] are rather unconvincing as the proposal(s) can be quite fittingly covered with the "constant reference" option of our hypothetical "switched" solution, did not investigate the quite obvious weakness of false alarms and/or escapes, only performed simulation that verified the intended function, and committed other design flaws such as creating elbows and bends in shield wires that allows reroute attacks. So far, we have yet to see a sound shield design that can claim this feat. Therefore, in this study we focus on detection method based on complete cuts.

As was discussed previously in Sect. 5.3.2, a reroute attack is to create a reroute between identified equipotential points by circuit editing with FIB, so that the net would not become open when sections of the wires are removed [2]. This forces active shield designs to only use parallel wires of minimum spacing and widths [22], without bending or elbows. In this case, the best placement of center of the milling (i.e., least likely to result in a complete cut of a wire) by the attacker is to place it at the middle of the space between any two wires. Conversely, the designer need to ensure within a certain radius d_{eff} of the center of the milling, the milled hole is at least deep enough to cut open the two closest shield wires. If we further assume an aspect ratio of the FIB R_{FIB}, then these two conditions together create a

restriction of milling hole diameter d. Since aspect ratio of the FIB R_{FIB} is a ratio between milling hole diameter d and depth D (which, if we assume the designer knows where his possible targeted wires are, is a known variable), this requirement translates into a maximum aspect ratio of the FIB $R_{\text{FIB, max}}$ the hypothetical active shield can detect, provided the milling is performed perpendicular to the IC.

5.4.4 *Impact of Milling Angle upon Effect of Bypass Attack*

One interesting question here is whether attacker would benefit if instead of milling vertically, he mills at an angle, as shown in Fig. 5.9. This question is directly relevant to appraising the threat of bypass attack: If the answer to the question is no, that means conventional vertical milling is the best-case scenario for the attacker in terms of possibility to bypass the shield; in that case, the security of any given active shield design against bypass attack can be simply deduced with the aspect ratio of the FIB milling tool it is able to detect. Otherwise, the problem of protecting against bypass attack would become much more complicated. If we assume the attacker were able to mill at an angle $\theta \leq \frac{1}{2}\pi$ relative to the surface of the IC, then he would cut off wires within region d'_{eff} instead of d_{eff}. We may evaluate d'_{eff}, and then taking derivative of $\frac{d'_{\text{eff}}}{d_{\text{eff}}}$ and letting it equal to zero yields the minimum d'_{eff} and angles it is reached. If we further assume shield wire height/width ratio $(A/R) = 2.5$ as in [32] (ITRS uses 2.34 [9]), minimum d'_{eff} over d_{eff} yields results shown in Table 5.2 for typical FIB aspect ratio R_{FIB}. From the table we see that by milling at approximately 68–69° angle the attacker can effectively reduce the diameter of area by 8–12 %, making it easier to bypass the shield. Since bypass

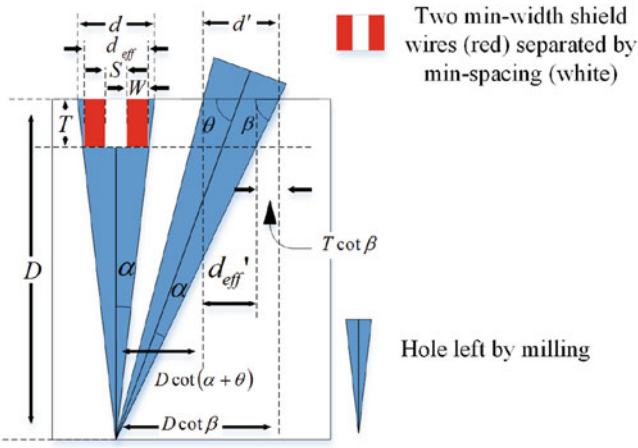


Fig. 5.9 Geometric calculations for non-perpendicular milling scenario

Table 5.2 Maximum achievable reduction of d_{eff} by milling at an angle

R_{FIB}	5	6	7	8	9	10
$\frac{d'_{\text{eff}}}{d_{\text{eff}}}$ (%)	92.12	90.58	89.47	88.63	87.98	87.45
θ_0 (°)	68.93	68.69	68.52	68.38	68.28	68.19

attack is considered a convenient and preferable approach [3], this suggests that it has the potential of becoming much more devastating. This result shows us that bypass attack is not a simple one dimensional issue, because obviously the benefit of milling at an optimized angle would be diminished if that angle would not agree with the aforementioned "milling center at middle between two wires" rule. Whether this reduction in milling diameter can be achieved will depend on the relative position of the shield wires with regard to targeted wires. In other words, this potential threat must be addressed by optimized positioning of the targeted wires, so that attacker cannot further reduce the possibility of complete cut by cleverly mill at an angle. This will require optimization based on layout information, which is best handled by a tool integrated with layout editors.

5.4.5 Algorithm to Find Exposed Area

We solve the problem of finding exposed area on targeted wires by first finding the complement area, i.e., the area that attacker wouldn't want to mill in. Consider wires above targeted wires in the layout. If the attacker completely cuts off a shield wire, he risks detection; even if it is merely a functional wire, he still risks corrupting information he wants to access without complete knowledge of its function through extensive reverse engineering. Using mathematical models shown in [33], complete cut will happen if center of milling exists within d_{faredge} from the far edge of the wire, where

$$d_{\text{faredge}} = \frac{D - H}{2R_{\text{FIB}}}$$

$$d_{\text{faredge}} = \frac{(2W + S)D}{2(2W + S)R_{\text{FIB}} + (A/R)W} \tag{5.1}$$

where d_{faredge} is the maximum distance from the center of the hole to the far side of the wire (shown in Fig. 5.10), where d is the diameter of the hole, W and S are minimum width and spacing of the shield layer, D is the depth of the hole, (A/R) is the aspect ratio of the shield wire metal, H is the thickness of the intersecting wire, and R_{FIB} is the aspect ratio given by the FIB technology the attacker is using. The aspect ratio represents the best FIB the shield will be able to defend against. Also, note that d_{faredge} represents the radius within which the milling will cut deep enough to completely cut open wires; therefore d_{faredge} is shorter than the radius ($\frac{1}{2}d$) of the hole radius left on the shield layer (as shown in Fig. 5.10a, b).

www.allitebooks.com

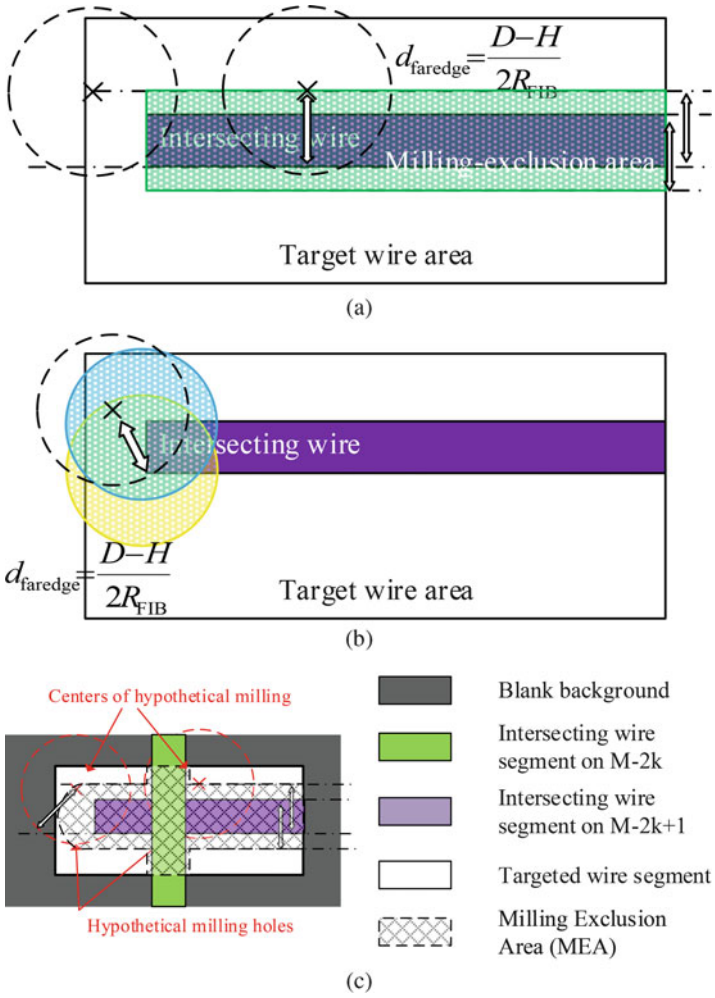


Fig. 5.10 Finding milling-exclusion area. (**a**) Milling-exclusion area on sides of intersecting wire; (**b**) Milling-exclusion area on ends of intersecting wire; (**c**) Complete milling-exclusion area in the presence of multiple intersecting wires

Equation 5.1 shows possibility to find the area which milling center should not fall inside. We term this area the *milling-exclusion area*. The desired *exposed area* will be its complement. Figure 5.10 shows how this area can be found for any given target wire and a wire on a higher layer capable of projecting this milling-exclusion area for it (henceforth termed as *intersecting wire*), assuming both are rectangular.

Boundaries of the milling-exclusion area can be found in two possible cases for a rectangular intersecting wire: the boundaries on the sides of the intersecting wire, and at both ends. The first kind is quite intuitive. As shown in Fig. 5.10a, the center of the milling cannot fall within d_{faredge} from the farther edge of the intersecting

wire, therefore boundaries of the first kind are two straight lines, each d_{faredge} away from the farther edge. The other kind of boundaries on ends are a bit more complex. Let's look at Fig. 5.10b. Consider the milling hole marked by the dotted circle. For it to precisely cut off the intersecting wire at each corner of the intersecting wire, its center must be on the edge of another circle centered at that corner, with same radius as itself. Any point within that other circle will still cut off that corner, although not necessarily the other corner. Therefore, the intersection area of both circles centered at both corners at an end constitute the complete set of milling center locations that will guarantee cut off both corners, i.e., a complete cut. Consequently, any rectangular intersecting wire will project a milling-exclusion area whose shape is the union of the shape shown in Fig. 5.10a, b.

Now, wires in layout designs are seldom rectangular, but they are always consisted of a number of rectangular wires, usually called *shapes* by layout design tools. Layout design tools, such as Synopsys IC Compiler, are able to provide sufficient information to determine each of these constituent rectangular wires in the form of their corner coordinates, with which a bit-map of mill-exclusion areas can be easily produced. By iterating through each of these constituent rectangular wires, mill-exclusion areas from each intersecting wire can be projected onto each wire that may carry sensitive information and become target of probing attack. This process is elaborated in the pseudocode as shown in Algorithm 5.1.

Algorithm 5.1: Proposed locator algorithm for exposed area.

```
   Input: targeted_nets, precision, all_layers
   Output: draw.script
1  begin
2      targeted_wire_shapes ⇐ get_net_shapes(targeted_nets)
3      N ⇐ sizeof_collection(targeted_wire_shapes)
4      for (i = 1:N) do
5          targeted_wire_shape ⇐ targeted_wire_shapes(i)
6          canvas_size ⇐ get_sizes(get_bounding_box(targeted_wire_shape))*precision
7          Print command in draw.script to create canvas in draw.script whose size equals to canvas_size
8          layers_above ⇐ get_layers_above(all_layers, get_layerof(targeted_wire_shape))
9          M ⇐ sizeof_collection(layers_above)
10         for (j = 1:M) do
11             this_layer ⇐ layers_above(j)
12             d_faredge_on_thislayer ⇐ (D−H)/(2R_FIB)
13             intersecting_wire_shapes ⇐ get_net_shapes(targeted_nets) in
               get_bounding_box(targeted_wire_shape) on this_layer
14             L ⇐ sizeof_collection(intersecting_wire_shapes)
15             for (k = 1:L) do
16                 intersecting_wire_shape ⇐ intersecting_wire_shapes(k)
17                 Print command in draw.script to create projection in draw.script whose radius/widths
                   equals to d_faredge_on_thislayer
18             end
19         end
20     end
21 end
```

As shown in Algorithm 5.1, the presented methodology starts with a set of logic nets. The algorithm first identifies their constituting wire shapes in

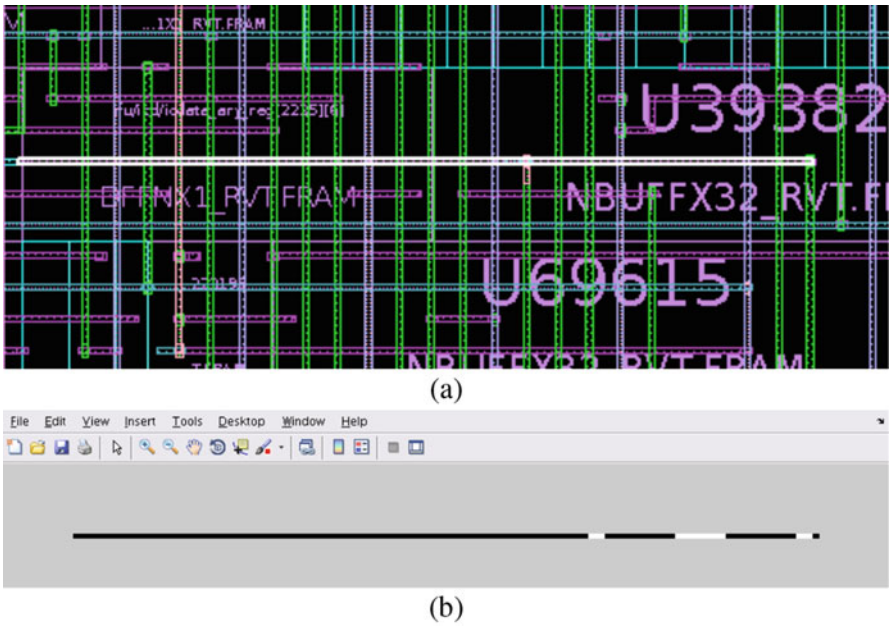


Fig. 5.11 Exemplary results produced by proposed algorithm. (**a**) Exemplary targeted wire (*highlighted*) in layout; (**b**) Mill-exclusion area (*black*) projected on canvas of same wire

targeted_wire_shapes. For each targeted wire shapes, a bitmap canvas is created, onto which mill-exclusion areas are to be projected once found. These coordinates are also given to the layout design tool to find *intersecting wire shapes* on each layer above. For each layer, a different d_{faredge} is calculated, which is then used for projections from all intersecting wire shapes on that layer. Coordinates of each intersecting wire shape are also retrieved to compute its mill-exclusion area, which is then projected to the aforementioned canvas (results shown in Fig. 5.11). Projection is done by locating ends and sides of each intersecting wire shape and print the corresponding projected mill-exclusion areas. After all mill-exclusion areas are projected, running the resulting script *draw.script* can easily determine existence and area of exposed area.

For processing efficiency and adaptability, both canvas creation and projection steps are stored by the layout design tool part of the algorithm in the format of MATLAB scripts. Considerations of probing attacks at non-perpendicular angles can also be included with simple modifications with trigonometric functions. Another possible concern is the precision of the bitmap method. The presented algorithm rounds towards minus infinity on borders, i.e., errors towards false positive. However, since mill-exclusion areas are convex, overlapping of mill-exclusion areas would unlikely cause the algorithm to declare a vulnerable point when there is none either.

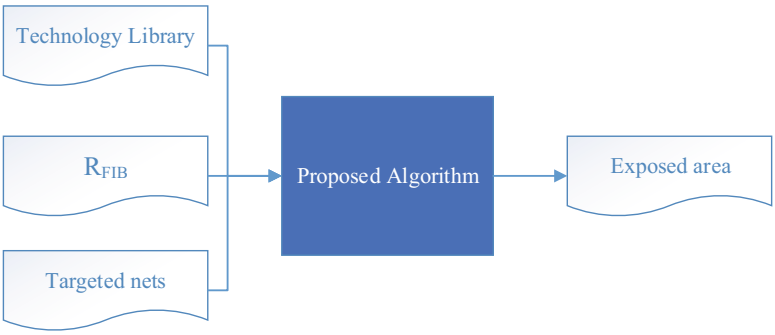


Fig. 5.12 A simplified diagram of presented algorithm to find exposed area

5.4.6 Discussions on Applications of Exposed Area Algorithm

Figure 5.12 shows a simplified diagram of Algorithm 5.1, where only inputs and output of the algorithm are shown. One apparent observation is that by controlling inputs to the algorithm, impacts of these inputs upon exposed area of targeted wires can be studied: for example, controlling R_{FIB} lets us study the threat of improved attacker sophistication, controlling targeted nets allows us to study benefits by relocating these wires, etc.

One important question is how the user should choose targeted nets to prepare against a serious attack. To start, it can be assumed that a designer would have a rough idea of which nets might attract most attention; and according to [30], linear combinations of these nets should also be considered. However, although it is always better safe than sorry, it might not be realistic to consider all possible candidates in this list, as it can be too many to process. Luckily, locating targeted wires remains a problem in published attacks [3]; and since we do not consider back-side attacks in this study, we can safely assume a minimum positioning error on the attacker, and rule out those nets that do not have a large enough area for attacker to reliably locate. Such an assumption along with assumption on attacker's R_{FIB} would constitute the quantified attacker capability model we advocated in Fig. 5.8a.

5.5 Conclusion

Probing attacks is defined as direct access of sensitive information by probing the physical wires that carries it. Reaching such wires necessitates milling, which is often satisfied by employing focused ion beam milling. Active shields detect milling by covering the layout with signal-carrying wires whose signals are monitored. Despite recent advances such as back-side attacks, their own limitations and lack of a better option have kept active shield as the most popular approach to

secure an IC against probing attacks. However, the active shield design is plagued with weaknesses that could be exploited by attackers. So far, existing research concentrated on securing the design against these weaknesses, at the cost of incurring high hardware cost and making themselves prohibitive to common victims to probing attacks such as smartcards and security tokens. In this chapter, we have reviewed existing probing attacks and antiprobing research, and presented a layout-driven framework to assess designs for vulnerabilities to probing attacks. Based on design principles and assessment rules we have established considering reported successful probing attacks, we presented an algorithm to analyze layout designs for potential vulnerabilities to probing attacks. We expect work presented here to serve as a basis for future methodologies to protect against probing attacks that are more effective, require lower hardware cost and applicable to a wider variety of ICs.

References

1. S. Skorobogatov, Physical attacks on tamper resistance: progress and lessons, in *Proceedings of 2nd ARO Special Workshop on Hardware Assurance*, Washington (2011)
2. C. Tarnovsky, Tarnovsky deconstruct processor, Youtube (2013) [Online]. Available: https://www.youtube.com/watch?v=w7PT0nrK2BE
3. V. Ray, Freud applications of fib: invasive fib attacks and countermeasures in hardware security devices, in *East-Coast Focused Ion Beam User Group Meeting* (2009)
4. R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems* (Wiley, New York, 2001)
5. WIRED, How to reverse-engineer a satellite tv smart card (2008) [Online]. Available: https://youtu.be/tnY7UVyaFiQ
6. K. Zetter, From the eye of a legal storm, murdoch's satellite-tv hacker tells all (2008) [Online]. Available: http://www.wired.com/2008/05/tarnovsky/
7. Invasive attacks (2014) [Online]. Available: https://www.sec.ei.tum.de/en/research/invasive-attacks/
8. I. Huber, F. Arthur, J.M. Scott, The role and nature of anti-tamper techniques in us defense acquisition, DTIC Document, Tech. Rep. (1999)
9. International Technology Roadmap for Semiconductors, 2013 edn., Interconnect (2013). [Online]. Available: http://www.itrs2.net/2013-itrs.html
10. X. Zhuang, T. Zhang, H.-H.S. Lee, S. Pande, Hardware assisted control flow obfuscation for embedded processors, in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. Ser. CASES '04 (ACM, New York, 2004), pp. 292–302. [Online]. Available: http://doi.acm.org/10.1145/1023833.1023873
11. R.S. Chakraborty, S. Bhunia, Harpoon: an obfuscation-based soc design methodology for hardware protection. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **28**(10), 1493–1502 (2009)
12. S.E. Quadir, J. Chen, D. Forte, N. Asadizanjani, S. Shahbazmohamadi, L. Wang, J. Chandy, M. Tehranipoor, A survey on chip to system reverse engineering. ACM J. Emerg. Technol. Comput. Syst. **13**(1), 6 (2016)
13. V. Sidorkin, E. van Veldhoven, E. van der Drift, P. Alkemade, H. Salemink, D. Maas, Sub-10-nm nanolithography with a scanning helium beam. J. Vac. Sci. Technol. B **27**(4), L18–L20 (2009)

14. Y. Fu, K.A.B. Ngoi, Investigation of aspect ratio of hole drilling from micro to nanoscale via focused ion beam fine milling, *Proceedings of The 5th Singapore-MIT Alliance Annual Symposium*. http://web.mit.edu/sma/about/overview/annualreports/AR-2004-2005/research/research06imst10.html
15. H. Wu, D. Ferranti, L. Stern, Precise nanofabrication with multiple ion beams for advanced circuit edit. Microelectron. Reliab. **54**(9), 1779–1784 (2014)
16. H. Wu, L. Stern, D. Xia, D. Ferranti, B. Thompson, K. Klein, C. Gonzalez, P. Rack, Focused helium ion beam deposited low resistivity cobalt metal lines with 10 nm resolution: implications for advanced circuit editing. J. Mater. Sci. Mater. Electron. **25**(2), 587–595 (2014)
17. C. Helfmeier, D. Nedospasov, C. Tarnovsky, J.S. Krissler, C. Boit, J.-P. Seifert, Breaking and entering through the silicon, in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (ACM, New York, 2013), pp. 733–744
18. C. Boit, C. Helfmeier, U. Kerst, Security risks posed by modern ic debug and diagnosis tools, in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)* (IEEE, Washington, 2013), pp. 3–11
19. A. Schlösser, D. Nedospasov, J. Kramer, S. Orlic, J.-P. Seifert, Simple photonic emission analysis of aes, in *Cryptographic hardware and embedded systems–CHES 2012* (Springer, Heidelberg, 2012), pp. 41–57
20. C. Boit, Fundamentals of photon emission (PEM) in silicon - electroluminescence for analysis of electronics circuit and device functionality, in *Microelectronics Failure Analysis* (ASM International, New York, 2004), pp. 356–368
21. C. Boit, R. Schlangen, U. Kerst, T. Lundquist, Physical techniques for chip-backside ic debug in nanotechnologies. IEEE Des. Test Comput. **3**, 250–257 (2008)
22. J.-M. Cioranesco, J.-L. Danger, T. Graba, S. Guilley, Y. Mathieu, D. Naccache, X.T. Ngo, Cryptographically secure shields, in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (IEEE, Arlington, 2014), pp. 25–31
23. Y. Xie, C. Bao, C. Serafy, T. Lu, A. Srivastava, M. Tehranipoor, Security and vulnerability implications of 3D ICs. IEEE Trans. Multiscale Comput. Syst. **2**(2), 108–122 (2016)
24. P. Laackmann, H. Taddiken, Apparatus for protecting an integrated circuit formed in a substrate and method for protecting the circuit against reverse engineering, 28 September 2004, US Patent 6,798,234
25. M. Ling, L. Wu, X. Li, X. Zhang, J. Hou, Y. Wang, Design of monitor and protect circuits against fib attack on chip security, in *2012 Eighth International Conference on Computational Intelligence and Security (CIS)* (IEEE, Guangzhou, 2012), pp. 530–533
26. A. Beit-Grogger, J. Riegebauer, Integrated circuit having an active shield, 8 November 2005, US Patent 6,962,294. [Online]. Available: https://www.google.com/patents/US6962294
27. S. Briais, J.-M. Cioranesco, J.-L. Danger, S. Guilley, D. Naccache, T. Porteboeuf, Random active shield, in *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)* (IEEE, Leuven, 2012), pp. 103–113
28. Invia., Active Shield IP (digital IP protecting System-on-Chip (SoC) against tampering through a metal mesh sensor) (2016) [Online]. Available: http://invia.fr/detectors/active-shield.aspx
29. F. Ungar, G. Schmid, Semiconductor chip with fib protection, 2 May 2006, US Patent 7,038,307. [Online]. Available: https://www.google.com/patents/US7038307
30. L. Wei, J. Zhang, F. Yuan, Y. Liu, J. Fan, Q. Xu, Vulnerability analysis for crypto devices against probing attack, in *2015 20th Asia and South Pacific Design Automation Conference (ASP-DAC)* (IEEE, Tokyo, 2015), pp. 827–832
31. C. Tarnovsky, Security failures in secure devices, in *Black Hat Briefings* (2008)
32. Freepdk45: Metal layers (2007) [Online]. Available: http://www.eda.ncsu.edu/wiki/FreePDK45:Metal_Layers
33. Q. Shi, N. Asadizanjani, D. Forte, M.M. Tehranipoor, A layout-driven framework to assess vulnerability of ics to microprobing attacks, in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2016)

34. S. Manich, M.S. Wamser, G. Sigl, Detection of probing attempts in secure ICs, in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (IEEE, San Francisco, 2012), pp. 134–139
35. Y. Ishai, A. Sahai, D. Wagner, Private circuits: securing hardware against probing attacks, in *Advances in Cryptology-CRYPTO 2003* (Springer, Heidelberg, 2003), pp. 463–481
36. M. Rivain, E. Prouff, Provably secure higher-order masking of aes, in *Cryptographic Hardware and Embedded Systems, CHES 2010* (Springer, Heidelberg, 2010), pp. 413–427
37. D.B. Roy, S. Bhasin, S. Guilley, J.-L. Danger, D. Mukhopadhyay, From theory to practice of private circuit: a cautionary note, in *2015 33rd IEEE International Conference on Computer Design (ICCD)* (IEEE, Washington, 2015), pp. 296–303
38. D. T. Ltd., Known attacks against smartcards (2015) [Online]. Available: http://www.infosecwriters.com/text_resources/pdf/Known_Attacks_Against_Smartcards.pdf
39. T. Xia, On-chip timing measurement. Ph.D. dissertation, University of Rhode Island (2003)

Chapter 6
Testing of Side-Channel Leakage of Cryptographic Intellectual Properties: Metrics and Evaluations

Debapriya Basu Roy, Shivam Bhasin, Sikhar Patranabis, and Debdeep Mukhopadhyay

6.1 Introduction

Semiconductor industry has really flourished under the intellectual property (IP) based business model, where proven IPs are widely reused to meet complexity and time constraints. Since security is emerging as the latest design parameter, along with performance, area, and power consumption, these IPs must be hardened for security. However, it is not always possible to modify each proven IP and add security features. To overcome this issue, required security targets are met at the system level by embedding cryptographic IPs in the system on chip (SoC). These cryptographic IPs ideally encrypt and decrypt all the sensitive data that is communicated among various IPs on the SoC and thus protect it from potential adversaries. The underlying cryptographic algorithms are rigorously tested against any theoretical weaknesses and often standardized.

D.B. Roy (✉) • S. Patranabis
Secured Embedded Architecture Laboratory (SEAL), Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, India
e-mail: deb.basu.roy@cse.iitkgp.ernet.in; dbroy24@gmail.com; sikhar.patranabis@cse.iitkgp.ernet.in

S. Bhasin
Temasek Laboratories NTU, Singapore, Singapore

Embedding Security and Privacy (ESP) IIT Kharagpur, Kharagpur, India
e-mail: sbhasin@ntu.edu.sg

D. Mukhopadhyay
Secured Embedded Architecture Laboratory (SEAL) Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, India

Embedding Security and Privacy (ESP) IIT Kharagpur, Kharagpur, India
e-mail: debdeep@cse.iitkgp.ernet.in

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_6

Since the used algorithms are provably secure, adversaries try to exploit other vulnerabilities. A commonly known vulnerability is side-channel attack (SCA [1]). SCA exploits unintentional physical leakage from semiconductor circuits in the form of power consumption, electromagnetic emanation, sound, timing, temperature, etc. The most commonly used physical channel is power consumption. It derives its basic from the behavior of a basic CMOS cell. It is well known that a CMOS gate draws non-negligible current whenever there is a transition ($1 \rightarrow 0$ or $0 \rightarrow 1$) on the input. When the input remains unchanged ($0 \rightarrow 0$ or $1 \rightarrow 1$), the current drawn is negligible or zero. Therefore just by observing the power consumption of a single CMOS gate, an adversary can retrieve some information about the inputs. Since the security of whole SoC relies on the embedded cryptographic IP, it should be tested and protected against SCA. SCA are now also tested by certification labs following Common Criteria [2] or FIPS [3] standards.

The simplest and the most common way to test for side-channel leakage is to perform an attack. The attack is based on intuition or characterization of the leakage model of the target device or IP. The actual power consumption is statistically compared against a set of key hypothesis, under the chosen leakage model. The efficiency of the attack can then be measured in terms of standard metrics like success rate and guessing entropy [4]. However, the attack directly relies on correctness of the leakage model and the quality of acquired side-channel measurements. To overcome these dependencies, alternative techniques like leakage detection and leakage assessment are used. Statistical test can also affect the efficiency of the attack but it was previously shown that all applied statistical tests are asymptotically equivalent [5].

Leakage detection is a preprocessing step in SCA testing. It serves two basic purpose. The first and the most obvious is quality assessment of the acquired side-channel measurement. In other words, leakage detection test checks if the collected measurement carries any relevant information as a quality check. This is even more important when the measurements are not synchronized and the adversary is not sure to capture good section of the measurement. The other and an important application of leakage detection is discovering zone of leakage or relevant point-of-interest (PoI). A side-channel trace or measurement can easily have millions of samples and the number of measurements can also grow in millions for protected implementations. To process such amount of data, one needs immense computing power. Therefore it is important to select the relevant PoI to improve the attack's efficiency.

Leakage assessment, on the other hand, is a generic evaluation technique. Normal SCA testing is highly dependent on the attack parameters (leakage model, statistical tool, and measurements). For evaluation and certification purpose, this scenario is not ideal as the number of parameters keep on increasing, which hinder the evaluation process to be comprehensive. To overcome this problem, evaluators rely on leakage assessment. These leakage assessment techniques are primarily global and parameter independent. It tests for any potential side-channel leakage in the

acquired measurement, without actually performing the attack. If a minimum level of leakage is found, the target is rendered insecure.

This chapter focusses on various aspect of side-channel testing on cryptographic IPs. It aims at establishing basic concepts of side-channel attack, leakage detection, and leakage assessment. Side-channel attack is the simplest and straightforward technique for testing cryptographic IC. In this context, we provide a formal description of an SCA along with its evaluation metrics, i.e., success rate and guessing entropy. Thereafter we describe the notion of leakage detection with a specific leakage detection tool known as normalized inter-class variance (NICV [6]). Relationship of NICV with standard parameters like signal-to-noise ratio (SNR), correlation, and other detection technique is also discussed. Practical case studies on hardware and software platforms are included to help readers understand the basic concepts. Lastly, leakage assessment is discussed, in particular test vector leakage assessment (TVLA [7]). Along with the basic concepts of TVLA, we derive relationship between TVLA and NICV. TVLA is also supported with a practical case study with direct application to SCA protected implementation.

The rest of the chapter is organized as follows. Section 6.2 discusses general background on statistical testing and hypothesis testing. This is followed with a formal description of side-channel attack with emphasis on evaluation metric, i.e., success rate and guessing entropy in Sect. 6.3. Leakage detection methods like NICV and SNR are dealt in Sect. 6.4 along with practical case studies on AES implementation. Section 6.5 describes leakage assessment methodology, in particular TVLA. The relationship between NICV and TVLA is derived in Sect. 6.6 followed by its extension to higher statistical orders in Sect. 6.7. and practical case study in Sect. 6.8. Finally conclusions are derived in Sect. 6.9.

6.2 Preliminaries on Statistical Testing and Testing of Hypothesis

Attack based evaluation strategies are difficult in the practical world as the number of attacks have been steadily increasing. Hence, it is desirable to have a black-box approach, the objective of which will be to quantify *any* side-channel information leakage through the power consumption of the device. The objective of the test strategy is thus to detect whether there is any information leakage, and not to precisely quantify how much of the information leakage is exploitable. Thus the purpose is to develop an *estimation* technique, which will rely on the *Theory of Statistical Hypothesis* and *Theory of Estimation*. The objective of this subsection is to provide a quick overview on the topic.

6.2.1 Sampling and Estimation

Sampling denotes the selection of a part of the aggregate statistical material with a view to obtaining information of the whole. This totality of statistical information on a particular character of all the members relevant to an investigation is called population. The selected part which is used to ascertain the characteristics of population is called *Sample*. The main objective of a good sampling is to obtain maximum information about the population with minimum effort, and to state the limits of accuracy of estimates based on samples.

Any statistical measure calculated on the basis of sample observations is called a *Statistic*, e.g., sample mean, sample variance. On the other hand, statistical measures based on the entire population are called a *Parameter*, e.g., population mean, population variance. The sample depends on chance, and hence the value of a statistic will vary from sample to sample, while the parameter remains a constant. The probability distribution of a statistic is called *sampling distribution*, and the standard deviation of the sampling distribution is called *standard error*, denoted as SE.

In the sequel, we assume simple random sampling, which implies that in the process of selecting a sample, every unit of the population has an equal probability of being included. Furthermore, we consider that the sampling is such that the probability of selection of any particular member remains constant throughout the selection process, irrespective of the fact that the member has been selected earlier or not. Such a sampling can be obtained by performing a simple random sampling with replacement (SRSWR), and is commonly called as simple sampling. It may be mentioned here that when the population size is infinite, even performing a simple random sampling without replacement (SRSWOR) will also result in simple sampling. Thus in cases where the population size is extremely large (say the plaintext pool in case of a 128 bit block cipher) both SRSWR and SRSWOR will result in simple sampling. Thus, we can assume that each sample members has the same probability distribution as the variable x in the population. Hence, the expectation $\mathbb{E}[x_i] = \mu$, where μ is the population mean. Likewise, the variance $\mathsf{Var}[x_i] = \mathbb{E}\left[(x_i - \mu)^2\right] = \sigma^2$, which is the population variance.

It can be proved that standard error for the mean, $\mathrm{SE}(\overline{(x)}) = \frac{\sigma}{\sqrt{n}}$, where n is the size of the sample. We state formally a subsequent result on standard errors which will be useful to understand the subsequent discussion on the detection test.

Theorem 1. *Consider two independent simple samples of sizes n_1 and n_2, with means $\overline{x_1}$ and $\overline{x_2}$, and standard deviations σ_1 and σ_2, respectively, then:*

$$\mathrm{SE}(\overline{x_1} - \overline{x_2}) = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}} \tag{6.1}$$

Different probability distributions are used in sampling theory, and they are all derived from the Normal distribution. They are (1) Standard Normal Distribution,

(2) Chi-square (ψ^2) Distribution, (3) Student's t Distribution, and (4) Snedecor's F Distribution.

In the following we provide a quick refresher to the Standard Normal Distribution, and the Student's t Distribution which shall be useful for understanding the remaining part of the sequel and in introducing the T-test.

6.2.2 Some Statistical Distributions

If a random variable x is normally distributed with mean μ and standard deviation σ, then the variable $z = \frac{x-\mu}{\sigma}$ is called a standard normal variate. The probability distribution of z is called Standard Normal Distribution, and is defined by the probability density function (pdf), $p(z) = (1/\sqrt{2\pi})e^{-z^2/2}$, where $-\infty < z < +\infty$.

An important result is if $\overline{x}$ denotes the mean of a random sample of size n, drawn from a normal population with mean μ and standard deviation (s.d.) σ, then, $z = \frac{\overline{x}-\mu}{\sigma/\sqrt{n}}$ follows standard normal distribution.

Likewise, a random variable is said to follow Student's t-distribution, or simply t-distribution, if its pdf is of the form: $f(t) = K(1 + \frac{t^2}{n})^{-(n+1)/1}$, where K is a constant and $-\infty < t < +\infty$. The parameter n is called the number of degrees of freedom (df.).

In statistics, the number of degrees of freedom is the number of values in the final calculation of a statistic that are free to vary. Estimates of statistical parameters can be based upon different amounts of information or data. The number of independent pieces of information that go into the estimate of a parameter is called the degrees of freedom. In general, the degrees of freedom of an estimate of a parameter are equal to the number of independent scores that go into the estimate minus the number of parameters used as intermediate steps in the estimation of the parameter itself (i.e., the sample variance has $N - 1$ degrees of freedom, since it is computed from N random scores minus the only one parameter estimated as intermediate step, which is the sample mean).

6.2.3 Estimation and Test of Significance

The objective of sampling is to infer the features of the population on the basis of sample observations. Statistical inference has two different ways: (1) Point Estimation and (2) Interval Estimation. In point estimation, the estimated value is given by a single quantity, which is a function of the given observations. In interval estimation, an interval within which the parameter is expected to lie is given by using two quantities based on sample values. This is known as Confidence Interval, and the two quantities which are used to specify the interval are known as Confidence Limits.

Let $x_1, x_2, \ldots, x_n$ be a random sample from a population of a known mathematical form which involves an unknown parameter θ. The confidence intervals specify two functions t_1 and t_2 based on sample observations such that the probability of θ being included in the interval (t_1, t_2) has a given value, say c: i.e., $P(t_1 \leq \theta \leq t_2) = c$. The probability c with which the confidence interval will include the true value of the parameter is known as Confidence Coefficient of the interval.

Let us illustrate this using an example. Let us consider a random sample of size n from a Normal population $N(\mu, \sigma^2)$, where the variance σ^2 is known. It is required to find confidence intervals for the mean, μ. We know that the sample mean $\overline{x}$ follows approximately a normal distribution with mean μ and variance σ^2/n. Thus, the statistic $z = (\overline{x} - \mu)/(\sigma/\sqrt{n})$ has a standard normal distribution. From the properties of the standard normal curve, 95 % of the area under the standard normal curve lies between the ordinates at $z = \pm 1.96$. Thus, we have

$$P[1.96 \leq (\overline{x} - \mu)/(\sigma/\sqrt{n}) \leq 1.96] = 0.95 \tag{6.2}$$

Thus arranging terms, the interval $(\overline{x} - 1.96\frac{\sigma}{\sqrt{n}}, \overline{x} + 1.96\frac{\sigma}{\sqrt{n}})$ is known as the 95 % confidence interval for μ. For almost sure limits, we replace the value 1.96 by the value 3.

In some cases, the population may not be truly a normal distribution, but the sample distributions of statistics based on large samples are approximately normal.

6.2.4 Test of Significance: Statistical Hypothesis Testing

Statistical tests often require to make decisions about a statistical population on the basis of sample observations. For example, given a random sample, it may be required to decide whether the population from which the sample has been obtained is a normal distribution with a specific mean and standard deviation. Any statement or assertion about a statistical population or its parameters is called a Statistical Hypothesis. The procedure which enables us to decide whether a certain hypothesis is true or not is called Test of Significance or Statistical Hypothesis Testing.

A statistical hypothesis which is set up (i.e., assumed) and whose validity is tested for possible rejection on the basis of sample observations is called Null Hypothesis. It is denoted as H_0 and tested for acceptance or rejection. On the other hand, an Alternative Hypothesis is a statistical hypothesis which differs from the null hypothesis, and is denoted as H_1. This hypothesis is not tested, its acceptance (or rejection) depends on the rejection (or acceptance) of that of the null hypothesis. For example, the null hypothesis may be that the population mean is 40, denoted as $H_0(\mu = 40)$. The alternative hypothesis could be $H_1(\mu \neq 40)$.

The sample is then analyzed to decide whether to reject or accept the null hypothesis. For this purpose, a suitable statistic, called Test Statistic is chosen. Its sampling distribution is determined, assuming that the null hypothesis is true.

The observed value of the statistic would be in general different from the expected value because of sampling fluctuations. However if the difference is very large, then the null hypothesis is rejected, Whereas, if the differences is less than a tolerable limit, then H_0 is not rejected. Thus it is necessary to formally determine these limits.

Assuming the null hypothesis to be true, the probability of obtaining a difference equal to or greater than the observed difference is computed. If this probability is found to be small, say less than 0.05, the conclusion is that the observed value of the statistic is rather unusual, and has arisen because the underlying assumption, i.e., the null hypothesis is not true. We say that the observed difference is significant at 5 % level of significance, and hence the null hypothesis is rejected at 5 % level of significance. The level of significance, say α also corresponds to a $(1 - \alpha)$ level of confidence. If, however, this probability is not very small, say more than 0.05, the observed difference cannot be considered unusual and is attributed to sampling fluctuations only. The difference now is not significant at 5 % level of significance.

The probability is inferred from the sampling distribution of the statistic. We find from the sampling distribution of the statistic the maximum difference which is exceeded say 5 % of cases. If the observed difference is larger than this value, the null hypothesis is rejected. If it is less, there is no reason to reject the null hypothesis.

Let us illustrate this with an example. Suppose, the sampling distribution of the statistic is a normal distribution. Since, the area under the normal curve under the ordinates at mean $\pm$ 1.96 (standard deviation) is only 5 %, the probability that the observed value of the statistic differs from the expected value by 1.96 times or more the standard error of the statistic (which is the standard deviation of the sampling distribution of the statistic) is 0.05. The probability of a larger difference will be still smaller.

If, therefore the statistic $z = \frac{\text{(Observed Value)}-\text{(Expected Value)}}{\text{Standard Error (SE)}}$ is either greater than 1.96 or less than -1.96, the null hypothesis H_0 is rejected at 5 % level of significance. The set of values $z \geq 1.96$, or $z \leq -1.96$ constitutes what is called the Critical Region of the test.

Thus the steps in Test of Significance can be summarized as follows:

1. Set up the Null Hypothesis H_0 and the Alternative Hypothesis, H_1. The null hypothesis usually specifies some parameter of the population: $H_0(\theta = \theta_0)$.
2. State the appropriate test statistic T and also its sampling distribution, when the null hypothesis is true. In large sample tests, the statistic $z = (T - \theta_0)/\text{SE}(T)$, which approximately follows standard Normal distribution, is often used. In small sample tests, the population is assumed to be normal and various test statistics are used which follow Standard Normal, Chi-Square, t distribution.
3. Select the level of significance, α of the test, or equivalently $(1 - \alpha)$ as the level of confidence.
4. Determine the critical region of the test for the chosen level of significance.
5. Compute the value of the test statistic z on the basis of sample data and the null hypothesis.
6. Check whether the computed value of the test statistic lies in the critical region. If it lies, then reject H_0, else H_0 is not rejected.

With this background, we eventually arrive at the proposed Test Vector Leakage assessment test, which is essentially a test of equality of two moments drawn independently and randomly from two populations. The starting point is the first moment, where equality of two means from the two samples are tested for equality. Thus, the statistic $T = \overline{x_1} - \overline{x_2}$, and thus the statistic $z = \frac{\overline{x_1} - \overline{x_2}}{\mathrm{SE}(\overline{x_1} - \overline{x_2})}$ is chosen for testing the null hypothesis: $H_0(\mu_1 = \mu_2)$, where μ_1 and μ_2 are the two means for the two independent samples. As discussed, the standard error of the difference of means $\overline{x_1} - \overline{x_2}$ is $\mathrm{SE}(\overline{x_1} - \overline{x_2}) = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$.

For a large distribution, the test statistic $z = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$ follows standard normal distribution. However, for tests with any sample sizes, a more exact sampling distribution for z is the t-distribution, and this gives rise to the Welch's t-test. The statistic z then follows the t-distribution with degrees of freedom calculated according to Welch–Satterthwaite, as $v = \frac{\mathrm{SE}(\overline{x_1} - \overline{x_2})}{\frac{(\sigma_1^2/n_1)}{n_1 - 1} + \frac{(\sigma_2^2/n_2)}{n_2 - 1}}$. The null hypothesis of two equal means is rejected when the test statistic $|z|$ exceeds a threshold of 4.5, which ensures with degrees of freedom > 1000, $P[|z| > 4.5] < 0.00001$, this threshold leads to a confidence of 0.99999.

6.3 Formalizing SCA and the Success Rate of Side-Channel Adversary: Guessing Entropy

In order to evaluate the several attack methods and also to compare the several cryptographic designs wrt these attacks several formal metrics have been developed. It is important to have an understanding of these security metrics which are based on formalizing these side-channel analysis techniques. In this section, we provide an overview on some of the fundamental metrics.

6.3.1 Success Rate of a Side-Channel Adversary

Side-channel adversaries work on a divide and conquer strategy, where the key space is divided into several equivalent classes. The attack is normally unable to distinguish keys which belong to the same class or partition, but is capable to distinguish between two keys which fall in different partitions. Thus one can formalize the adversary as an algorithm which targets a given implementation of a cryptographic algorithm, formally denoted as E_K, where K denotes the key space. The adversary also assumes a leakage model for the key denoted as L. The leakage model provides some information of the key or some other desirable information. Also the adversary is bounded in terms of computational resources and thus the adversary $A_{E_K,L}$ is an algorithm with time complexity τ, memory complexity m,

Algorithm 6.1: Formal definition of success rate of a side-channel attack

Input: K, L, $\mathsf{E_K}$
Output: 0 (failure), 1 (success)

$k \in_R K$
$s = \gamma(k)$
$\mathbf{g} = [g_1, \ldots, g_o] \leftarrow A_{E_k,L}$
if $s \in g$ **then**
  return 0
else
  return 1

making q queries to the target implementation of the cryptographic algorithm. Note that the leakage function is not capable of distinguishing certain keys, hence inducing a partition S on the entire key space K. The mapping from K to S can be captured by a function γ such that $s = \gamma(k)$, where $s \in S$ and $k \in K$, and $|S| << |K|$. The objective of the adversary is to determine the corresponding equivalence class to which a chosen key k belongs denoted as $s = \gamma(k)$ with a non-negligible probability.

As an analogy consider the Hamming Weight or Hamming-Distance leakage model, which divides the entire key space into equivalence classes or partitions, which the attacker tries to distinguish with sufficient observations. The output of the adversary is based on the ciphertext (black-box information) and the leakage (side-channel information) outputs a *guess vector*, which are the key classes sorted in terms of descending order of being a likely candidate. We thus define an order-o ($o \leq |S|$) adversary when the adversary produces the guessing vector as $\mathsf{g} = [g_1, \ldots, g_o]$, where g_1 is the most likely candidate, and so on. Now having defined the adversary let us try to understand formally the side-channel experiment to define the metrics.

More precisely, the side-channel attack is defined as an experiment $\mathrm{Exp}_{\mathsf{A_{E_K,L}}}$ where $A_{E_K,L}$ is an adversary with time complexity τ, memory complexity m, and making q queries to the target implementation of the cryptographic algorithm. In the experiment for any k chosen randomly from K, when the adversary $A_{E_k,L}$ outputs the guessing vector g, the attack is considered as a success if the corresponding key class denoted as $s = \gamma(k)$ is such that $s \in g$. More formally the experiment for a side-channel attack of order-o is as in Algorithm 6.1. The experiment returns 0 or 1 indicating a success or failure of the attack.

The oth order success rate of the side-channel attack $\mathsf{A_{E_K,L}}$ against the key classes or partitions S is defined as:

$$\mathbf{Succ}^{\mathbf{o}}_{\mathbf{A_{E_K,L}}}(\tau, m, k) = \mathsf{Pr}[\mathrm{Exp}_{A_{E_K,L}} = 1]$$

Algorithm 6.2: Formal definition of guessing entropy

Input: K, L, E_{K}
Output: Key class i

1 $k \in_R K$
2 $s = \gamma(k)$
3 $\mathbf{g} = [g_1, \ldots, g_o] \leftarrow A_{E_k,L}$
4 **return** i such that $g_i = s$

6.3.2 Guessing Entropy of an Adversary

The above metric for an oth order attack implies the success rate for an attack where the remaining workload is o-key classes. Thus the attacker has a maximum of o-key classes to which the required k may belong. While the above definition for a given order is fixed wrt the remaining work load, the following definition of guessing entropy provides a more flexible definition for the remaining work load. It measures the average number of key candidates to test after the attack. We formally state the definition, based on the adversaries experiment as in Algorithm 6.2.

The Guessing Entropy of the adversary $A_{E_k,L}$ against a key class variable S is defined as:

$$\mathbf{GE}_{\mathbf{A}_{\mathbf{E}_\mathbf{K},\mathbf{L}}}(\tau, m, k) = \mathsf{E}[\mathrm{Exp}_{\mathbf{A}_{\mathbf{E}_\mathbf{K},\mathbf{L}}}]$$

6.4 Leakage Detection in SCA Traces: NICV and SNR

A formal framework for side-channel analysis and the methodology to evaluate its success was discussed in the previous section. An effective SCA stands on three parameters: measurements, leakage model, and distinguisher. If any of these parameters are sub-optimal, then it has a direct impact on SCA and its success. SCA countermeasures are also designed on the same principle such that it is hard to estimate the optimal leakage model, distinguisher, or acquire quality measurements. This section focuses on assessing the quality of side-channel measurements. A practical problem in SCA is that the measured traces can be huge with multimillion sample points. Processing traces of such size requires time and computing power. In order to optimize the attack, it is important to identify a small set of the so-called zone of leakage or relevant point-of-interest (PoI). This process of finding PoI is known as leakage detection. Moreover as a first test, leakage detection also informs if the traces carry any relevant information or not, thus aiding in assessing the quality of measurements. In the following, we provide the theoretical background and rationale to normalized inter-class variance (NICV) as a leakage detection tool. The relation of NICV to signal-to-noise ratio (SNR) is also discussed followed by case study on practical implementations of AES.

6.4.1 Normalized Inter-Class Variance

Normalized inter-class variance (NICV) is a technique which is designed to detect relevant PoI in an SCA trace [6]. Detection of PoI is used to compress the trace which results in acceleration of SCA. NICV can be computed based on public parameters like plaintext or ciphertext and doesn't need a profiling or pre-characterization of the target device. Moreover, NICV is leakage model agnostic which means that it can be applied without the knowledge of the target implementation. These properties make NICV an ideal candidate for accessing quality of measurements. The technical background of NICV is discussed in the following.

A side-channel adversary acquired leakage measurement $Y \in \mathbb{R}$ corresponding to a public parameter X (let's say a byte of plaintext or ciphertext, i.e., $\mathcal{X} = \mathbb{F}_2^8$). In general, Y can be continuous, but X must be discrete (and $\mathcal{X}$ must be of finite cardinality). Then, for all leakage prediction function L of the leakage knowing the value of x taken by X (as per Proposition 5 in [8]), we have

$$\rho^2\left[L(X);Y\right] = \underbrace{\rho^2\left[L(X);\mathbb{E}\left[Y|X\right]\right]}_{0\leq\,\cdot\,\leq 1} \times \rho^2\left[\mathbb{E}\left[Y|X\right];Y\right] \quad . \tag{6.3}$$

Here, $\mathbb{E}$ and Var denote the expectation and the variance, respectively, whereas ρ represents correlation. Equation (6.3) was further simplified in Corollary 8 of [8] to derive:

$$\rho^2\left[\mathbb{E}\left[Y|X\right];Y\right] = \frac{\mathsf{Var}\left[\mathbb{E}\left[Y|X\right]\right]}{\mathsf{Var}\left[Y\right]} \quad , \tag{6.4}$$

The term in Eq. (6.4) is further called as the *normalized inter-class variance* (NICV). It is an ANOVA (ANalysis Of VAriance) F-test, as a ratio between the explained variance and the total variance.

From Eqs. (6.3) and (6.4), we can derive that for all prediction function $L : \mathbb{F}_2^8 \to \mathbb{R}$:

$$0 \leq \rho^2\left[L(X);Y\right] \leq \frac{\mathsf{Var}\left[\mathbb{E}\left[Y|X\right]\right]}{\mathsf{Var}\left[Y\right]} = \text{NICV} \leq 1 \quad . \tag{6.5}$$

Thus, NICV is the *envelop* or maximum of all possible correlations computable from X with Y or NICV denotes the best-case results. Although Eq. (6.5) contains an equality, however, it is almost impossible to achieve in practical scenario. The equality can occur if and only if $L(x) = \mathbb{E}\left[Y|X = x\right]$, i.e., L is the optimal prediction function. The difference can come from various practical reasons like:

– The adversary knows the exact prediction function, but not the correct key. For instance, let us assume the traces can be written as $Y = w_H(S(X \oplus k^\star)) + N$, where $k^\star \in \mathbb{F}_2^8$ is the correct key, $S : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ is a substitution box, w_H is

the Hamming weight function, and N is some measurement noise, that typically follows a centered normal distribution $N \sim \mathcal{N}(0, \sigma^2)$. In this case, the optimal prediction function $L(x) = \mathbb{E}[Y|X = x]$ is equal to: $L(x) = w_H(S(X \oplus k^\star))$ (the only hypothesis on the noise is that it is *centered* and *mixed additively* with the sensitive variable). This argument is at the base of the soundness of CPA: $\forall k \neq k^\star, \rho[w_H(S(X \oplus k)); Y] \leq \rho[w_H(S(X \oplus k^\star)); Y] \leq \sqrt{\mathsf{Var}[\mathbb{E}[Y|X]] / \mathsf{Var}[Y]}$.

– The leakage model is incorrect, for instance, $L(x) = w_H(x \oplus k^\star)$, when $Y = w_H(S(x \oplus k^\star)) + N$.
– The leakage model is a close approximation of the actual leakage model, for instance, $L(x) = w_H(S(x \oplus k^\star))$, whereas actually $Y = \sum_{i=1}^{8} \beta_i \cdot S_i(x \oplus k^\star) + N$, where $\beta_i \approx 1$, but slightly deviate from one.

The distance between CPA and NICV, in non-information theoretic attacks (i.e., attacks in the proportional / ordinal scale, as opposed to the nominal scale [9]) is similar to the distance between perceived information (PI) and mutual information (MI) [10].

In practice, the (square) CPA value does not attain the NICV value, owing to noise and other imperfections. This is illustrated in Fig. 6.1.

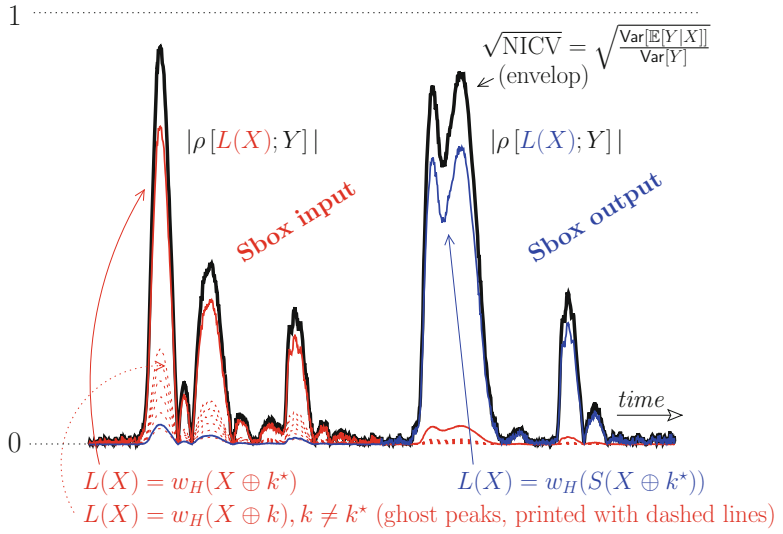


Fig. 6.1 NICV metric when $Y = \sum_{i=1}^{8} \beta_i \cdot S_i(x \oplus k^\star) + N$, and attack results for some prediction functions

6.4.2 NICV and SNR

NICV can be extended to establish relationship with signal-to-noise ratio (SNR), which is a widely used metric across domains. In the case of SCA, the signal refers to that part of the measurement which is directly related to sensitive data manipulation. When power measurements are considered, the signal is the current drawn for sensitive data computation. For instance, the measurement $Y = w_H(S(X \oplus k^\star)) + N$. The signal here directly refers to computation related to sensitive key $k^\star$ only. Note that a typical cryptographic algorithm will have several key bytes manipulated independently (16 for AES). The current drawn by computation related to other key bytes except $k^\star$ is considered noise and known as algorithmic noise. The noise N is the sum of algorithmic noise and measurement noise.

Now we derive the relation between NICV and SNR. It should be noted that we are considering only a byte of the ciphertext which has 2^8 number of possible values. If the public parameter X is uniformly distributed, the NICV in itself *is not* a distinguisher. Indeed, if we assume that $Y = L(X) + N = w_H(S(X \oplus k^\star)) + N$, then:

$$\begin{aligned}
\mathsf{Var}\left[\mathbb{E}\left[Y|X\right]\right] &= \textstyle\sum_{x\in\mathcal{X}} \mathsf{P}[X = x]\mathbb{E}\left[Y|X = x\right]^2 - \mathbb{E}\left[Y\right]^2 \\
&= \tfrac{1}{2^8} \textstyle\sum_{x\in\mathcal{X}} \mathbb{E}\left[w_H(S(x \oplus k^\star)) + N\right]^2 \\
&\quad - \left(\textstyle\sum_{x\in\mathcal{X}} \mathbb{E}\left[w_H(S(x \oplus k^\star)) + N\right]\right)^2 \\
&= \tfrac{1}{2^8} \textstyle\sum_{x'=x\oplus k^\star\in\mathcal{X}} \mathbb{E}\left[w_H(S(x'))\right]^2 \\
&\quad - \left(\textstyle\sum_{x'=x\oplus k^\star\in\mathcal{X}} \mathbb{E}\left[w_H(S(x'))\right]\right)^2 \\
&= \mathsf{Var}\left[w_H(S(X))\right].
\end{aligned}$$

Further elaborating denominator of NICV, $\mathsf{Var}\left[Y\right] = \mathsf{Var}\left[w_H(S(X))\right] + \mathsf{Var}\left[N\right]$.

To put both the equations together, we get

$$\boxed{\text{NICV} = \frac{\mathsf{Var}\left[\mathbb{E}\left[Y|X\right]\right]}{\mathsf{Var}\left[Y\right]} = \frac{1}{1 + \frac{1}{\text{SNR}}}} \,, \tag{6.6}$$

where the SNR is the ratio between:

- the signal, i.e., the variance of the informative part, namely $\mathsf{Var}\left[w_H(S(X \oplus k^\star))\right]$, and
- the noise, considered as the variance $\mathsf{Var}\left[N\right]$. An approximate estimation of noise is the intra-class variance, i.e., $\mathbb{E}\left[\mathsf{Var}\left[Y|X\right]\right]$.

Clearly, Eq. (6.6) does not depend on the secret key $k^\star$ as both Y and X are public parameters known to the attacker. Moreover, this expression is free from the leakage

model $L(X)$, which means that NICV does not depend on the implementation. Thus, NICV searches for all linear dependencies of public parameter X with available leakage traces Y independent of the implementation.

There are several advantages of computing NICV as compared to SNR. The main reason for using NICV over SNR is that, NICV is a bounded quantity. As shown in Eq. (6.5), the value of NICV lies in the range [0,1]. Thus the quality of the trace is measured directly as a percentage. Owing to this property, it is possible to compare the quality of measurements directly.

Next, the computation of SNR requires much more traces than NICV. The noise component $\mathsf{Var}[N]$ of SNR is mainly composed of the intra-class variance $\mathbb{E}[\mathsf{Var}[Y|X]]$. In order to properly estimate $\mathbb{E}[\mathsf{Var}[Y|X]]$, the adversary must acquire at least two measurements per class in order to compute the variance. Of course, it is desirable to have much more traces per class to compute the noise component of SNR. Since NICV depends on global variance of the measurements, it takes fewer traces to estimate.

6.4.3 Related Work in Leakage Detection

Initial methods for leakage detection were based on templates [11]. In this method, PoI are those which maximizes $\sum_{i,j=1}^{n}(T_i - T_j)$, for T templates for n subkey values. T_i is the average of the traces when the sensitive variable belongs to the class i. It was improved to Sum Of Squared pairwise Differences (SOSD [12]) as $\sum_{i,j=1}^{n}(T_i - T_j)^2$, to avoid cancellation of leakage of opposite polarity. SOSD was then extended to Sum Of Squared pairwise T-differences (SOST [12]) by normalizing it with variances as $\sum_{i,j=1}^{n}\left(\frac{T_i - T_j}{\sqrt{\frac{\sigma_i^2}{m_i} + \frac{\sigma_j^2}{m_j}}}\right)^2$. Here σ_i is the variance of T in class i, and m_i is the number of samples in class i. However, template based method needs an access to clones of a device.

6.4.4 Case Study: Application on AES

The main application of NICV is to find the interesting time samples for accelerating SCA. An SCA trace can easily have millions of points, however, only few of these points qualify as PoI. It is of interest of the adversary or evaluator to detect these PoI and compress the final measurement.

The NICV metric is first tested on a software implementation of AES-256 running on an ATMEL AVR microcontroller. A standard trace of this implementation contains seven million points and needs roughly 5.3 Mbytes of disk space when stored in the most compressed format. These details are in respect to a LeCroy wavescanner 6100A oscilloscope with a bandwidth of 1 GHz. The NICV is then

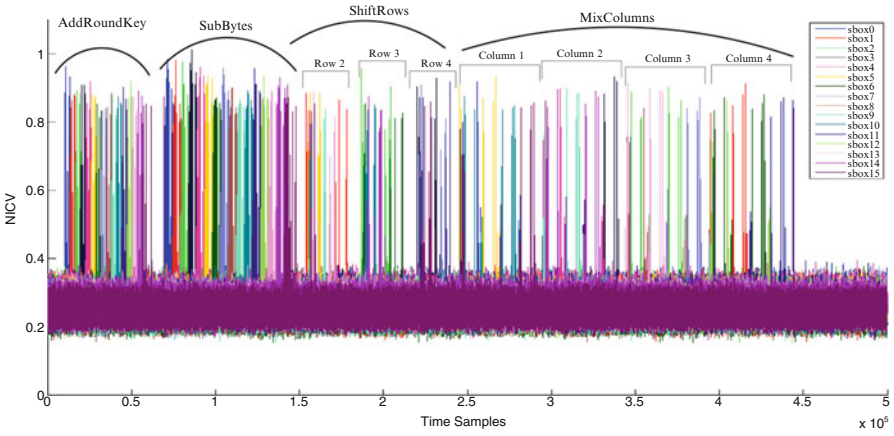


Fig. 6.2 NICV computed for an AES-128 software implementation to detect each round operation

applied to identify operations related to each byte of sensitive key $k_0^\star - k_{15}{}^\star$. Figure 6.2 shows the computation of NICV on the first round. The computations of $k_0^\star$ for round 1 take only $\approx$ 1000 time samples. Since each $k^\star$ is associated with a different substitution box (Sbox), the bytes are labeled as the Sbox number, i.e., Sbox0 to Sbox15. Once the interesting time samples corresponding to each executed operation is known, the trace size is compressed from 7,000,000 to 1000, i.e., a gain of roughly 7000×. NICV also enables us to reverse engineer the underlying software implementation. We computed NICV for all the 16 bytes of the plaintext and plotted the 16 NICV curves in Fig. 6.2 (depicted in different colors). By closely observing Fig. 6.2, we can distinguish individual operations from the sequence of byte execution. Each NICV curve (each color) shows all sensitive leakages related to that particular byte. Moreover, with a little knowledge of the algorithm, one can easily follow the execution of the algorithm. For example, the execution of all the bytes in a particular sequence indicates the SubBytes or AddRoundKey operation. Manipulation of bytes in sequence $\{1, 5, 9, 13\}, \{2, 6, 10, 14\}$, and $\{3, 7, 11, 15\}$ indicates the ShiftRows operations. The ShiftRows operation of AES shifts circularly 3 out of 4 rows with different constant. This can be clearly seen in Fig. 6.2: only three rows are manipulated and the bytes in the first row, i.e., $\{0, 4, 8, 12\}$ are not used during this time. Similarly MixColumns can also be identified by just looking at the bytes manipulated together.

Another common problem in SCA is the choice of leakage model which directly affects the efficiency of the attack. As shown in Sect. 6.4.1, the square of the correlation between modeled leakage ($L(X, K)$) and traces ($Y = L(X, K^\star) + N$) is smaller or equal to NICV, where N represents a noise. The equality exists only if the modeled leakage is the same as the traces. A real implementation can have different active components at different point of time. For example, the activity corresponding to Sbox input computation will be at different time than Sbox output computation.

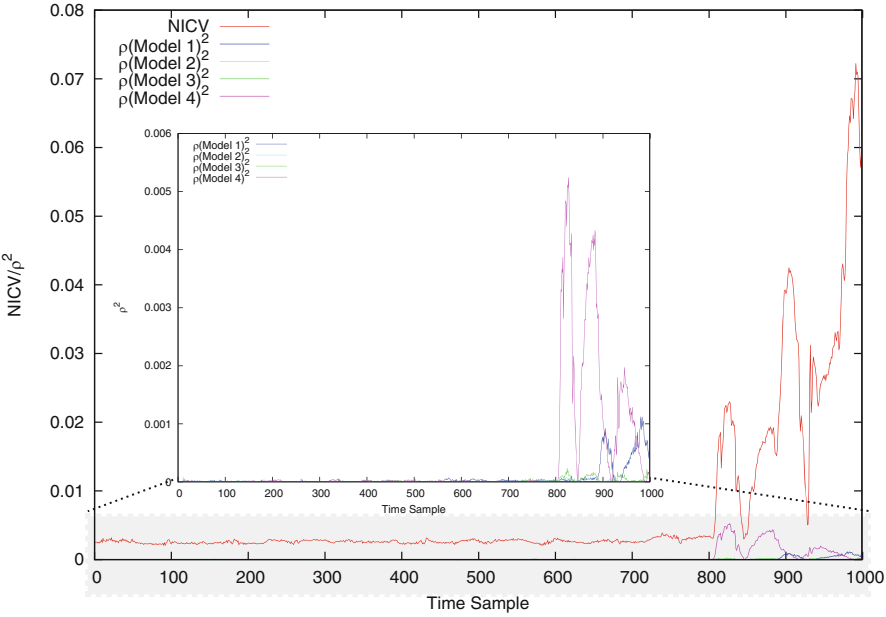


Fig. 6.3 NICV vs ρ^2 of four different models

If we test the implementation against these two models, leakage corresponding to each model shall be visible on the measurement at different times.

We tested the model on an AES-128 hardware implementation using two different leakage models on an FPGA and acquired SCA traces. The first leakage model corresponds to the state register present before the Sbox operation of AES, i.e., $w_H(\text{val}_i \oplus \text{val}_f) \in [\![0, 8]\!]$ (Model 1) and $\text{val}_i \oplus val_f \in [\![0, 255]\!]$ (Model 2). w_H is the Hamming weight function. Similar models are built for another register which is intentionally introduced at the output of the Sbox, i.e., $w_H(S(\text{val}_i) \oplus S(\text{val}_f)) \in [\![0, 8]\!]$ (Model 3) and $S(\text{val}_i) \oplus S(\text{val}_f) \in [\![0, 255]\!]$ (Model 4). Figure 6.3 shows the square of correlation of four different leakage models with the traces against the NICV curve. It can be simply inferred from Fig. 6.3 that Model 4 performs the best while Model 2 is the worst. The gap between NICV and $\rho(\text{Model } 4)^2$ is quite large due to reasons mentioned in Sect. 6.4.1. Thus NICV depicts leakage corresponding to several leakage model in the measurements.

6.5 Test Vector Leakage Assessment Methodology

The previous section discussed NICV as a leakage detection technique. Leakage detection is a pre-processing step in the side-channel analysis, which helps in locating PoI and thus improving attack efficiency. However, it does not provide

conclusive output about security of the IP. In order to evaluate the security, one must rely on leakage assessment.

When an IP has to be certified for security evaluation, a certification lab must conduct series of SCA on the IP under test. The list must be updated regularly to keep track of state-of-the-art vulnerabilities. These attacks might involve applying different distinguishers like correlation, mutual information or linear regression [13], etc. Moreover, several leakage models must be tested. The range of attacks and leakage model makes certification a time-consuming and non-comprehensive task. Moreover, it can be error prone as it highly depends on choice of distinguisher and model.

The aforementioned shortcomings motivate the need of a generic testing method which could effectively evaluate the security of the IP. Leakage assessment is a methodology which checks for potential side-channel leakage in the target without actually performing the attack. The methodology is based on the fact that SCA exploits side-channel information corresponding to sensitive intermediate values. This presence of side-channel information can be statistically detected using hypothesis testing. The seminal work which proposed this methodology named it Test Vector Leakage Assessment (TVLA [7]).

TVLA partitions the side-channel measurements of traces Y on the basis on plaintext. In order to compute TVLA, one must acquire two sets of traces. While one set corresponds to a fixed key and fixed plaintext as input to the cryptographic IP, the second set collects traces corresponding to fixed key and random plaintext. Thereafter a hypothesis testing performed by assuming a null hypothesis that the two sets of traces have identical means and variance. Background on hypothesis testing was previously discussed in Sect. 6.2. If the null hypothesis is accepted, it signifies that the traces carry no sensitive information. On the other hand, a rejected null hypothesis indicates presence of exploitable leakage. In [7], Welch's t-test is used for statistical testing and the technique is called non-specific t-test. This can be expressed as:

$$\text{TVLA} = \frac{\mathbb{E}\left[Y_r\right] - \mathbb{E}\left[Y_f\right]}{\sqrt{\frac{\text{Var}[Y_r]}{m_r} + \frac{\text{Var}[Y_f]}{m_f}}} = \frac{\mu_r - \mu_f}{\sqrt{\frac{\sigma_r^2}{m_r} + \frac{\sigma_f^2}{m_f}}}\ , \tag{6.7}$$

where Y_r and Y_f correspond to set of traces with random and fixed plaintext, respectively. m_r, m_f signifies the number of traces in set Y_r, Y_f, respectively. The mean and standard deviation of set Y_r is denoted by μ_r and σ_r. Similarly, μ_f and σ_f refer to mean and standard deviation of Y_f. The TVLA value must be contained with the range $\pm R$, to accept the null hypothesis. If the TVLA value exceeds the absolute threshold R in either polarity, the device is considered to leak sensitive side-channel information. The intuitive value generally used is $R = 4.5$, which for large $m(> 5000)$ accepts or rejects the null hypothesis with a confidence of 99.9999 %. So at any time, if the TVLA value of the target IP is either less than -4.5 or more than 4.5, it can be rendered as leaking side-channel information.

The non-specific *t*-test assess the leakage in the target IP without performing an attack and without any hypothesis on the underlying implementation. It finally provides a level of confidence on the presence of exploitable leakage from the target. However no information is provided about the source of leakage and methodology to exploit that leakage. Further on, the author proposes specific *t*-test testing, which is partitioned based on sensitive intermediate value rather than public plaintext, to potentially find the source of leakage.

TVLA was originally demonstrated on first order leakage in a univariate setting [7]. This work was further extended to a multivariate setting in [14] along with performance enhancement for efficient computation of TVLA. Another aspect of *T*-test based assessment was explored in [15]. While [7, 14] used an unpaired *T*-test, the use of paired *T*-test was proposed in [15]. The main argument for switching to paired *T*-test (TVLA_P) is that it is less sensitive to environmental noise and gradual temperature changes. It can be expressed as:

$$\text{TVLA}_P = \frac{\mu_d}{\sqrt{\frac{\sigma_d^2}{m}}} \ , \tag{6.8}$$

μ_d and σ_d are mean and standard deviation of paired difference of Y_r and Y_f. The paired *T*-test was also combined with a moving average based computation, leading to better results for multivariate setting. Nevertheless, the moving average improvement can also be applied to unpaired *T*-Test.

6.6 Equivalence of NICV and TVLA

TVLA and NICV were previously discussed in detail. In this section, we derive the relationship between these two metrics. This allows us to estimate one metric by knowledge of the other. TVLA is a simple *T-test*. Actually if we look closely, NICV is similar to statistical *F-test*. Now, in case of statistical *T-test*, we only consider two different classes whereas in case of statistical *F-test*, we consider multiple different classes. In the scenario of two class statistical *F-test*, the result of statistical *F-test* is found to be proportional to square of the statistical *T-test* result. This shows a clear relationship between NICV and TVLA. Moreover, this also exhibits relationship between TVLA and SNR as from NICV, we can easily calculate the SNR of the underlying attack setup. In the following, we will try to obtain the exact relationship between NICV and TVLA.

Let us assume that we have some side-channel traces which belong to two different classes: class 1 and class 2. Both the classes have same cardinality *N*. The mean of class 1 is μ_1 and mean of class 2 is μ_2. Now, the computation of NICV

is as follows:

$$\text{NICV} = \frac{\frac{1}{2}\sum_{i=1}^{2}(\mu_i - \mu)^2}{\frac{1}{2N}\sum_{i=1}^{2N}(x_i - \mu)^2} \tag{6.9}$$

Similarly we can compute TVLA^2 as follows:

$$\text{TVLA}^2 = \frac{(\mu_1 - \mu_2)^2}{\frac{V_1}{N} + \frac{V_2}{N}} = \frac{K}{\frac{V_1}{N} + \frac{V_2}{N}} \tag{6.10}$$

where V_1, V_2 are variance of class 1 and class 2, respectively, and $K = (\mu_1 - \mu_2)^2$. μ is the mean of the all side-channel traces and is equal to $\mu = \frac{\mu_1 + \mu_2}{2}$. Now we will consider only the numerator part of the NICV formulation which is

$$\begin{aligned}
&\frac{1}{2}\sum_{i=1}^{2}(\mu_i - \mu)^2 \\
&= \frac{1}{2}((\mu_1 - \mu)^2 + (\mu_2 - \mu)^2) \\
&= \frac{1}{2}\left(\left(\mu_1 - \frac{\mu_1 + \mu_2}{2}\right)^2 + \left(\mu_2 - \frac{\mu_1 + \mu_2}{2}\right)^2\right) \\
&= \frac{1}{4}(\mu_1 - \mu_2)^2 = \frac{K}{4}
\end{aligned} \tag{6.11}$$

Next we will consider the denominator part of the NICV computation which is as follows:

$$\begin{aligned}
&\frac{1}{2N}\sum_{i=1}^{i=2N}(x_i - \mu)^2 \\
&= \frac{1}{2N}\sum_{i=1}^{i=2N}(x_i - \frac{\mu_1 + \mu_2}{2})^2 \\
&= \frac{1}{2N}\sum_{i=1}^{i=2N}(x_i^2 - x_i(\mu_1 + \mu_2) + \frac{(\mu_1 + \mu_2)^2}{4}) \\
&= \frac{1}{2N}\sum_{i=1}^{i=2N}(x_i^2 - x_i(\mu_1 + \mu_2)) + \frac{(\mu_1 + \mu_2)^2}{4}
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2N}\sum_{x_i \in class1}(x_i^2 - x_i(\mu_1+\mu_2)) + \frac{1}{2N}\sum_{x_i \in class2}(x_i^2 - x_i(\mu_1+\mu_2)) + \frac{(\mu_1+\mu_2)^2}{4} \\
&= \frac{1}{2N}\sum_{x_i \in class1}((x_i-\mu_1)^2 - \mu_1^2 + \mu_1 x_i - \mu_2 x_i) + \\
&\quad \frac{1}{2N}\sum_{x_i \in class2}((x_i-\mu_2)^2 - \mu_2^2 + \mu_2 x_i - \mu_1 x_i) + \frac{(\mu_1+\mu_2)^2}{4} \\
&= \frac{1}{2N}\sum_{x_i \in class1}(x_i-\mu_1)^2 + \frac{1}{2N}\sum_{x_i \in class2}(x_i-\mu_2)^2 + \frac{1}{2N}\sum_{x_i \in class1} x_i(\mu_1-\mu_2) + \\
&\quad \frac{1}{2N}\sum_{x_i \in class2} x_i(\mu_2-\mu_1) + \frac{(\mu_1+\mu_2)^2}{4} - \frac{\mu_1^2}{2} - \frac{\mu_2^2}{2} \\
&= \frac{V_1}{2} + \frac{V_2}{2} + \frac{1}{2}(\mu_1-\mu_2)\mu_1 + \frac{1}{2}(\mu_2-\mu_1)\mu_2 + \frac{(\mu_1+\mu_2)^2}{4} - \frac{\mu_1^2}{2} - \frac{\mu_2^2}{2} \\
&= \frac{V_1}{2} + \frac{V_2}{2} + \frac{K}{4}
\end{aligned}
\tag{6.12}
$$

Hence the NICV value can be re-written as below

$$
\begin{aligned}
\text{NICV} &= \frac{\frac{K}{4}}{\frac{V_1}{2} + \frac{V_2}{2} + \frac{K}{4}} \\
&= \frac{1}{\frac{2(V_1+V_2)}{K} + 1} \\
&= \frac{1}{\frac{2N}{\text{TVLA}^2} + 1} \\
&\propto \text{TVLA}^2
\end{aligned}
\tag{6.13}
$$

This demonstrates that NICV value is directly proportional to square of TVLA value.

6.7 TVLA on Higher Order Side-Channel Attacks

In this section, we will focus on application of TVLA on higher order side channel analysis. The TVLA methodology, discussed in the previous sections is applicable to first order side-channel analysis where the adversary is allowed to exploit leakage of only one intermediate value. Using this methodology, we can analyze first order

side-channel vulnerability of any crypto-system. Additionally, we can also validate side-channel security of any countermeasure which claims to be resistant against first order attacks. However, any countermeasure which prevents first order attack can be vulnerable against higher order attacks. Hence, it is imperative to analyze these countermeasures against higher order attacks and for this we also need to update the formulation of TVLA metric computation. In [14], authors have proposed a detailed description of how to modify the TVLA metric for higher order attacks. In this section, we will give a brief overview of that formulation. Now, the difference between first order and higher order side-channel analysis is that in case of higher order analysis we need to pre-process the side-channel analysis. For example, in case of second order side-channel analysis, we need to convert the traces into mean free squared traces. To compute TVLA on higher order, we need to estimate mean and variance of these pre-processed side-channel traces.

6.7.1 Estimation of Mean

In this section, we will first describe the estimation of first parameter of TVLA for higher order analysis which is mean in-case of first order of analysis. For higher order analysis, we need to pre-process the side-channel traces and then need to compute the mean of pre-processed side-channel traces. In this subsection, our objective is to estimate the mean of the pre-processed side-channel traces for higher order analysis.

A dth order side-channel attack exploits leakage of d different intermediate variables and tries to get access to the secret key. To formulate the corresponding TVLA metric for dth order side-channel analysis we introduce the following notations:

1. M_d= dth order raw statistical moment of a random variable X. M_d is computed as follows:

$$M_d = \mathbb{E}\left[X^d\right] \tag{6.14}$$

   where $\mathbb{E}[.]$ is the expectation operator. For $d = 1$, $M_1 = \mu$ (mean).
2. CM_d= dth order central moment ($d > 1$) of random variable X. CM_d is computed as follows:

$$\mathrm{CM}_d = \mathbb{E}\left[(X - \mu)^d\right] \tag{6.15}$$

   For $d = 2$, CM_2= variance of the distribution.
3. SM_d= dth order standardized moment ($d > 2$) of random variable X. SM_d is computed as follows:

$$\mathrm{SM}_d = \mathbb{E}\left[\left(\frac{X - \mu}{\sigma}\right)^d\right] \tag{6.16}$$

   where σ is the standard deviation of the distribution.

In case of first order test, we use mean of the two different classes which is actually raw statistical moment of them. In case of second order, we will use CM_2 which is actually the variance of the side-channel traces. For third and higher order of analysis we will require SM_d of side-channel traces. Now, the most efficient way of calculating these statistical metrics is to use online one pass algorithms which compute the value of these metrics for each new trace, added to the distribution. This actually allows computation of these metrics during the acquisition of side-channel traces and saves storage as well as postprocessing time. There are various ways by which a user can compute the discussed statistical metrics in one pass fashion. But care should be taken to avoid any numerically unstable steps as that will introduce errors in the computation. A stable incremental one pass algorithm was proposed in [14] which we will describe next.

Let us assume that $M_{1,Q}$ denote the raw statistical moment of a given set Q. Additionally. let y be the new acquired side-channel trace added to the set Q. Let $M_{1,Q'}$ be the new raw statistical moment of the enlarged set Q'. This can be computed as follows:

$$M_{1,Q'} = M_{1,Q} + \frac{\Delta}{n} \tag{6.17}$$

where $\Delta = y - M_{1,Q}$ and n denotes the number of side-channel traces in enlarged set Q'. Similarly we can estimate central moment (CM_d) using one pass algorithm. However to do that, we need to estimate another metric central sum (CS_d) which can be computed as follows:

$$\mathrm{CS}_d = \sum (x_i - \mu)^d \tag{6.18}$$

From CS_d, we can calculate CM_d as follows:

$$\mathrm{CM}_d = \frac{\mathrm{CS}_d}{n} \tag{6.19}$$

The one pass incremental computation for CS_d is given below:

$$\mathrm{CS}_{d,Q'} = \mathrm{CS}_{d,Q} + \sum_{k=1}^{d-2} \binom{d}{k} \mathrm{CS}_{d-k,Q} \left(\frac{-\Delta}{n}\right)^k + \left(\frac{n-1 \times \Delta}{n}\right)^d \left[1 - \left(\frac{-1}{n-1}\right)^{d-1}\right] \tag{6.20}$$

Finally, we can compute SM_d by using the following formula:

$$\mathrm{SM}_d = \frac{\mathrm{CM}_d}{(\sqrt{(\mathrm{CM}_2)^d}} \tag{6.21}$$

Using the above described equations, we now have formulations for computing CM_d and SM_d in one pass. This allows us to estimate the mean of the pre-processed side-channel traces for higher order analysis. For second order, it is the variance of the side-channel traces. For higher order, we need to compute SM_d which will act as the mean of the pre-processed side-channel traces.

6.7.2 *Estimation of Variance*

In this subsection, our focus will be on the estimation of the variance of the pre-process side-channel traces. In case of second order analysis, the side-channel trace set Y needs to be converted into mean free squared, i.e., $(Y-\mu)^2$. Variance of this mean free squared traces can be calculated as follows:

$$\begin{aligned}
\mathsf{Var}\left[(Y-\mu)^2\right] &= \frac{1}{n}\sum\left((y-\mu)^2 - \frac{1}{n}\sum(y-\mu)^2\right)^2 \\
&= \frac{1}{n}\sum\left((y-\mu)^2 - \mathrm{CM}_2\right)^2 \\
&= \frac{1}{n}\sum(y-\mu)^4 - \frac{2}{n}\mathrm{CM}_2\sum(x-\mu)^2 + \mathrm{CM}_2^2 \\
&= \mathrm{CM}_4 - \mathrm{CM}_2^2
\end{aligned} \tag{6.22}$$

For higher order analysis (for example, dth order), where the traces are needed to be standardized, we first compute $Z = \left(\frac{Y-\mu}{\sigma}\right)^d$. The computation of variance of Z can be done as follows:

$$\begin{aligned}
\mathsf{Var}\left[Z^2\right] &= \mathrm{SM}_{2d} - \mathrm{SM}_d^2 \\
&= \frac{\mathrm{CM}_{2d} - \mathrm{CM}_d^2}{\mathrm{CM}_2^d}
\end{aligned} \tag{6.23}$$

Now, with this formulation we can now estimate mean and variance of pre-processed side-channel traces for any arbitrary order side-channel analysis and once we have them, we can compute TVLA very easily. These incremental one pass computations are not only beneficiary for saving storage and computation time, but also can be utilized for efficient computation of TVLA on chip [16]. For more detailed analysis on applying TVLA on higher order side-channel analysis, the readers are encouraged to read [14].

6.8 Case Study: Private Circuit

In the previous sections, we have discussed about different metrics which have been proposed for evaluation of side-channel vulnerability. We have also highlighted how two such metrics, TVLA and NICV, are actually equivalent, and can be used interchangeably. In this section, we will provide a case study where we employ TVLA to asses side-channel security of a side-channel secured crypto-implementation. Our case study will be on the seminal work [17] of Ishai, Sahai, and Wagner where they proposed a theoretical countermeasure against probing attack, which is the strongest form of SCA. Although the proposed countermeasure [17], here after referred to as *ISW scheme* or t-private circuits, is suited for probing attack, it can be used to prevent power or Electromagnetic SCA. Probing attack considers a strong adversary who is capable of observing exact values of one or several internal nets of the circuit including the nets containing sensitive information. In case of SCA, the adversary cannot observe the exact value of the sensitive information (key bits or key-dependent nets) but a linear or non-linear transformation of sensitive values like Hamming weight or Hamming-distance. Thus this countermeasure which prevents much stronger probing attack can also be used to prevent passive side-channel attacks like power or EM leakage based attack. Private circuits also form the basis of much studied countermeasures against SCA known as masking [18]. In particular, Rivain and Prouff [18] had proposed a dth order provably secure masking scheme for AES. This masking scheme was derived by optimizing the hardware-oriented t-private circuits for software implementations. The t-private circuits are secure against an adversary capable of observing any t nets of the circuit at any given time instant. Construction of t-private circuits involves $2t$ number of random bits for each input bit of the circuit. Moreover, it requires $2t + 1$ random bits for each 2-input *AND* gate present in the circuit. The overall complexity of the design is $\mathcal{O}(nt^2)$ where n is the number of gates in the circuit.

The countermeasure, proposed in [17] is based on sound theoretical proof but with some inherent assumptions which may not be valid in practical scenario. In this case study, we will try to identify the practical scenarios in which private circuit may fail to provide us the desired security. We would like to state that we are not making any claim against security of private circuit, but we are pointing out the dangerous situations where expected security can be compromised. For this we have implemented a lightweight block cipher *SIMON* [19] using private circuit methodology on SASEBO-GII board. As we evaluate private circuits in a pure hardware setting, we choose to use the original ISW scheme as presented in [17] to be fair towards various schemes branched out of it. Moreover, the implementation in [20] is primarily proposed for FPGA target. However, we intend to study the security of private circuits in general. The choice of block cipher is motivated by its compact design and simplistic construction using basic gates like *AND* and *XOR*.

Detailed case study on side-channel security of private circuits in practical scenarios can be found in [21]. The implemented private circuits are analyzed against SCA using EM traces and correlation power analysis [22]. We primarily

have implemented three different versions of *SIMON* on FPGA using private circuit, which are as follows:

- *Optimized SIMON*: In this case, SIMON is implemented according to the ISW scheme [17], but the design tool is free to optimize the circuit. As our implementation platform is a Virtex-5 FPGA which has six-input Look-up table (LUT), design tool (in our case Xilinx ISE) will optimize the circuit to reduce the resource requirement of the design. This is an example of *lazy engineering* approach, where designer is allowing the tool to do modification on the design without being aware of its possible impact on the security of private circuits.
- 2-*input LUT based SIMON*: Here, to mimic the private circuit methodology exactly on the FPGA, we have constrained the design tool to map each two-input gate to a single LUT. In other words, though an LUT has six inputs, it is modeled as two-input gate and gate-level optimization is minimized.
- *Synchronized 2-input LUT based SIMON*: This is nearly similar to the previous methodology. The only difference is that each gate or LUT is preceded and followed by flip-flops so that each and every input to the gates is synchronized and glitches are minimized (if not suppressed, see [23]).

We will show that among these three, *Optimized SIMON* can be broken using CPA whereas, 2-*input LUT based SIMON* is resistant against CPA, but fails *TVLA* test. Finally we will show that *Synchronized* 2-*input LUT based SIMON* is not only resistant against CPA, but also passes *TVLA* test [24, 25].

We could have based our study on more popular ciphers like AES or PRESENT, but with the FPGA in consideration, it was not possible to fit protected designs when using constraints *LOCK_PINS* and *KEEP*. In fact, we had to switch from SIMON64/96 to SIMON32/64, to be able to fit all the designs on the FPGA.

6.8.1 Experimental Analysis and Result

In this section, we perform practical evaluation of *private circuit* by executing SCA on block cipher *SIMON*. We start with the experimental setup, followed by the analysis of the obtained result on various implementation of private circuits.

To perform SCA, side-channel traces are acquired using an EM antenna of the HZ–15 kit from Langer, placed over a power decoupling capacitor. The traces are captured on a 54855 Infiniium Agilent oscilloscope at a sampling rate of 2 GSample/s. In the following subsections we will discuss the result of SCA on different variants of *t-private SIMON*.

6.8.1.1 Optimized SIMON

In this setting, we implemented SIMON using private circuit methodology with $t = 1$. At this stage, the FPGA tool was not constrained and was allowed to do

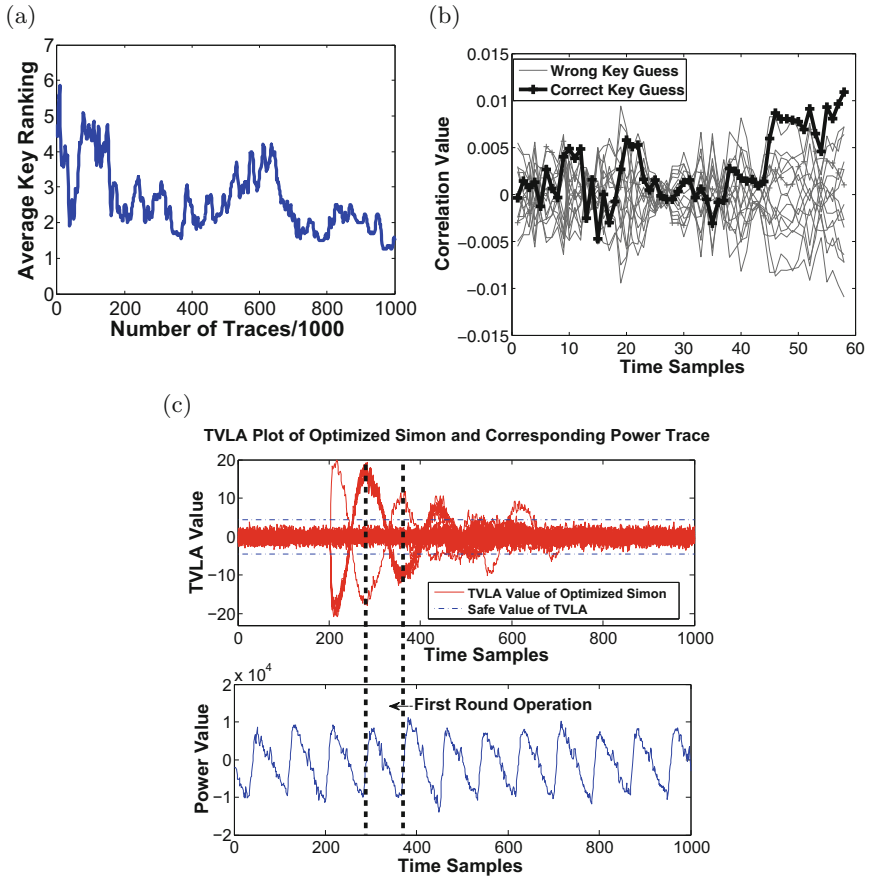


Fig. 6.4 Side-channel analysis of optimized SIMON. (**a**) Average key ranking. (**b**) Correlation Value of Key Guesses. (**c**) TVLA plot

all optimizations if possible. We collected 100,000 EM traces and performed *TVLA* test, along with *CPA*. *TVLA* test is performed on the first round output of *SIMON* for each bit of plaintext. Basically, the traces are partitioned into 2 sets on the basis of value of concerned bit of round output, i.e., 1 or 0. Thereafter, we compute the means (μ_1, μ_0) and standard deviations (σ_1, σ_0) of the two sets and we compute the corresponding TVLA value. The result can be seen in Fig. 6.4c. The first few peaks in the plot are due to the plaintext loading, however, after that it can be seen that for some bits, value of TVLA is much larger than the safe value of TVLA, i.e., ± 4.5 for secure system [24, 25]. Thus this implementation of SIMON is vulnerable to SCA though it is designed by private circuit. The leakage comes due to the optimizations applied by CAD tools. To validate our claim further, we carried out CPA around the sample points where TVLA peaks can be observed. We have chosen

Hamming-distance as our attack model and we targeted first round of *SIMON32/64*. The leakage model can be written as follows:

$$L_{\mathrm{HD}} = w_H(R(x_{i+1}, x_i) \oplus key_i \oplus x_{i+1}),$$

where x_i, x_{i+1} are parts of plaintext, key_i is the round key, and R is the round function. The first round operation of *SIMON32/64* involves 16 bits of key and we try to recover key nibble by nibble. Now, generally for a nibble, key space is 16; however, as SIMON has no non-linear operation on the key in first round, for each key guess there is another key candidate which has exactly same value of correlation coefficient with opposite polarity. Due to this symmetry, total key space reduces from 16 to 8. To measure the success rate, we have calculated *average key ranking* (or equivalently we can calculate guessing entropy) over all the nibbles at intervals of 1000 power traces and the corresponding result is shown in Fig. 6.4a. At the end of 100,000 traces, we have been able to recover the correct key for two nibbles and for the rest of the two nibbles, ranking of correct key is 2, which clearly shows successful attack. Figure 6.4b shows the correlation values of different key guesses. Though the gap between the correct key and wrong key guesses is marginal, it is consistent as indicated by Fig. 6.4b, providing us enough evidence to conclude the attack as successful. The reason for this small nearest rival distance is the Hamming-distance attack model which does not incorporate randomization of private circuit. The key ranking curve is not very smooth. Theoretically addition of traces should lower the key ranking, but the reverse phenomenon can happen in practical scenario, as it depends upon the quality of the acquired traces. CPA is a statistical analysis and this phenomenon is statistical artifact. Moreover, the average key ranking will depend on the combined progression of all the 4 nibbles. Nevertheless, due to the CAD tool optimization, it is possible to successfully retrieve secret information from private circuit (Fig. 6.5).

6.8.1.2 2-Input LUT Based SIMON

In the previous discussion, we provided experimental validations of the disastrous effect of CAD tool optimization on private circuits. Designer can use various attributes and constraints to disable the optimization property of CAD tools, and hence can mimic the private circuit more efficiently (when the input HDL file is described structurally with LUTs). In this implementation, we have constrained the Xilinx ISE tool to treat each LUT as a 2-input gate using `KEEP` and `LOCK_PIN` attributes. *SIMON 32/64* is then designed using this 2-input LUT gates so that no optimizations can be applied by the CAD tools. Similar SCA is carried out for this implementation also and corresponding result is shown in Fig. 6.6. As we can see, TVLA plot still shows occurrence of information leakage, though it is less significant compared to information leakage observed for *Optimized SIMON*. Average key ranking plot (Fig. 6.6b) also shows improvement in terms of more resistance against SCA. However, as there is still information leakage, as shown

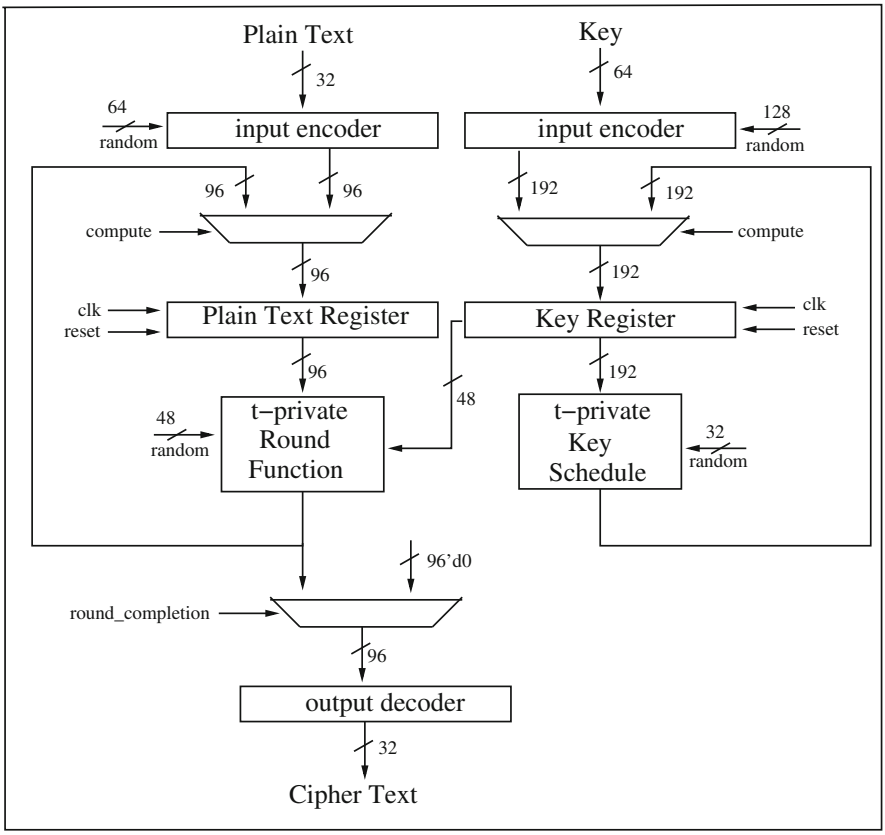


Fig. 6.5 Architectural overview of SIMON

in TVLA plot, the system could be broken by attack which employs better model. In other words, we cannot be absolutely confident about the side-channel resistance of the implementation.

6.8.1.3 Synchronized 2-Input LUT Based SIMON

In the previous discussion, we have shown how side-channel attack is resisted by 2-input LUT based SIMON. However, TVLA plot of 2-input LUT based SIMON still shows some information leakage. In this subsection, we tackle this issue.

It was shown in [21] how delay in random variables can disrupt the security of private circuit. In our implementation, random variables are generated using a maximum length LFSR and we suspect that asynchronous random variables as the reason of the information leakage. To analyze this, we have made a new

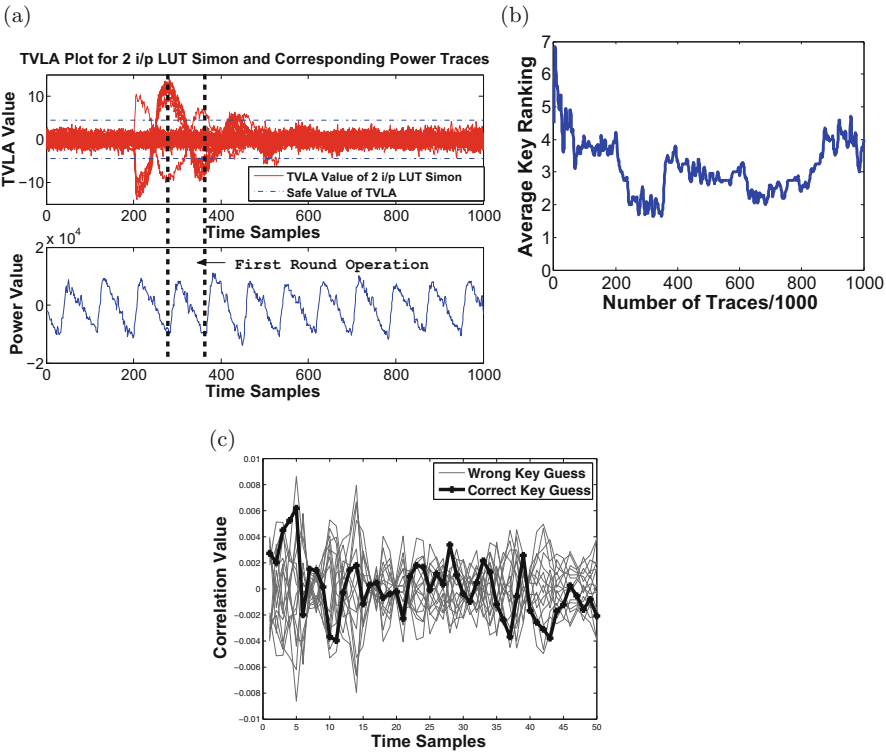


Fig. 6.6 Side-channel analysis of 2-input LUT SIMON. (**a**) TVLA plot. (**b**) Average key ranking. (**c**) Correlation value of key guesses

implementation where each 2-input LUT is followed by a flip-flop,[1] so that if there is any delay or glitches on random variables, it will be handled locally and will not propagate across the circuit. We have carried out similar side-channel analysis on this implementation and the result can be seen in Fig. 6.7. Now before analyzing the result we will like to state that this implementation has more clock cycle requirement compared to previous two implementations due to the presence of flip-flops after each 2-input LUT. For example, in the previous two implementations, first round operation occurs just after the start of encryption, whereas in this implementation first round output is obtained in the 9-th clock cycle after the start of encryption. As we are doing TVLA test on the first round output of SIMON, TVLA peak should appear near the clock cycle where first round output is obtained. As we can see, in TVLA plot (Fig. 6.7a), there are few peaks near the start of encryption

[1] In the *ISW* scheme [17], logic gates of *stateless circuits* are combinational instances, which, by design, evaluate as soon as one input changes. Our addition of the flip-flop after each combinational gate ensures that they evaluate only once, which is the legal mode of operation for *ISW* to be sound.

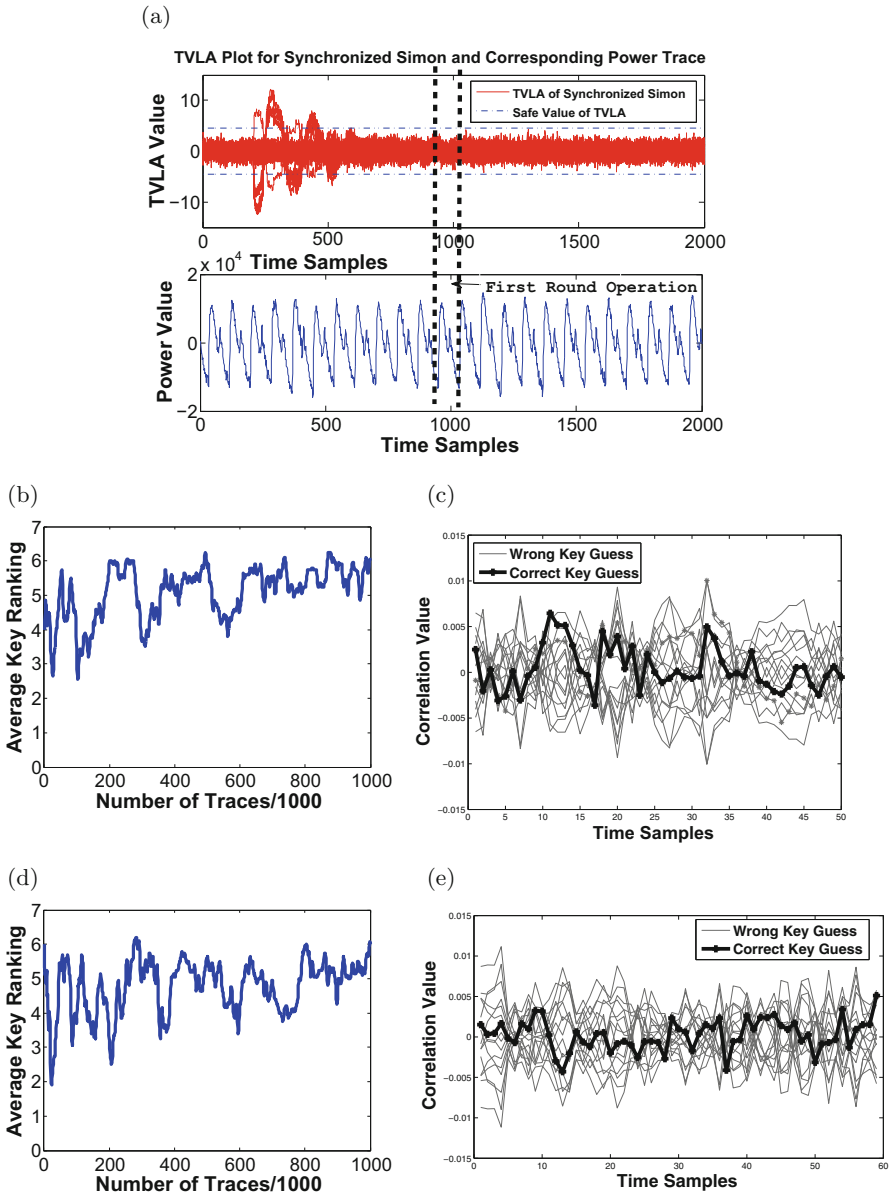


Fig. 6.7 Side-channel analysis of synchronized 2-input LUT SIMON. (**a**) TVLA plot. (**b**) Avg. key rank (TVLA peak). (**c**) Correlation value at TVLA peak. (**d**) Avg. key rank (first round). (**e**) Correlation value at first round

Table 6.1 Summary of side-channel analysis

Design name	TVLA test	Max. TVLA leakage	Avg. key ranking	Remarks
Optimized SIMON	Fails, significant information leakage	18	Key ranking is low, successful attack	Not secure
2-input LUT based SIMON	Fails, but less information leakage	12	Key ranking is high, attack fails	Secure against CPA with HD model
Synchronized 2-input LUT based SIMON	Passes: no leakage at first round	3.5	Key ranking is high, attack fails	Secure

which indicates plaintext loading, but no peak at the first round operation. However, to eliminate any ambiguity, we have performed CPA around both the first round operation and TVLA peak. Result shows that in both cases, side-channel attack is not working and implementation under attack can be considered secure against side-channel attack. Thus in this case, theoretically secure private circuit is translated into practically secure private circuit implementation. The summary of our experimental analysis is shown in Table 6.1.

6.9 Conclusion

This book chapter focuses on the basic concepts of side-channel vulnerability evaluation of cryptographic IPs. In this context we have presented three different metrics: Guessing Entropy, NICV, and TVLA. For each of the metric, we have first described the basic mathematical background, followed by example case studies. We have also highlighted how two different metrics TVLA and NICV are related to each other and how we can modify TVLA formulation to address higher order side-channel analysis. A thorough case study on popular side-channel countermeasure *private circuit* has also been presented where we analyze side-channel security of three different implementation strategies of *private circuits* using TVLA.

Acknowledgements We will like to thank Professor Sylvain Guilley from Telecom Paristech for his valuable inputs on side channel which greatly improved the chapter.

References

1. P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in *Annual International Cryptology Conference* (Springer, Berlin, 1999), pp. 388–397
2. Common Criteria consortium, Application of Attack Potential to Smartcards — v2.7, March (2009)

3. NIST FIPS (Federal Information Processing Standards) publication 140–3, Security requirements for cryptographic modules (Draft, Revised). 09/11 (2009), p. 63
4. F.-X. Standaert, T.G. Malkin, M. Yung, A unified framework for the analysis of side-channel key recovery attacks, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer, Berlin, 2009), pp. 443–461
5. S. Mangard, E. Oswald, F.-X. Standaert, One for all; all for one: unifying standard differential power analysis attacks. IET Inf. Secur. **5**(2), 100–110 (2011)
6. S. Bhasin, J.-L. Danger, S. Guilley, Z. Najm, Side-channel leakage and trace compression using normalized inter-class variance, in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy* (ACM, New York, 2014), p. 7
7. B.J.G. Goodwill, J. Jaffe, P. Rohatgi, et al, A testing methodology for side-channel resistance validation. NIST Non-invasive Attack Testing Workshop (2011)
8. E. Prouff, M. Rivain, R. Bevan, Statistical analysis of second order differential power analysis. IEEE Trans. Comput. **58**(6), 799–811 (2009)
9. C. Whitnall, E. Oswald, F.-X. Standaert, The myth of generic DPA and the magic of learning, in *Cryptographers Track at the RSA Conference* (Springer, Berlin, 2014), pp. 183–205
10. M. Renauld, F.-X. Standaert, N. Veyrat-Charvillon, D. Kamel, D. Flandre, A formal study of power variability issues and side-channel attacks for nanoscale devices, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer, Berlin, 2011), pp. 109–128
11. S. Chari, J.R Rao, P. Rohatgi, Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems* (Springer, Berlin, 2002), pp. 13–28
12. B. Gierlichs, K. Lemke-Rust, C. Paar, Templates vs. stochastic methods, in *International Workshop on Cryptographic Hardware and Embedded Systems* (Springer, Berlin, 2006), pp. 15–29
13. J. Doget, E. Prouff, M. Rivain, F.-X. Standaert, Univariate side channel attacks and leakage modeling. J. Cryptogr. Eng. **1**(2), 123–144 (2011)
14. T. Schneider, A. Moradi, Leakage assessment methodology. in *International Workshop on Cryptographic Hardware and Embedded Systems* (Springer, Berlin, 2015), pp. 495–513
15. A.A. Ding, C. Chen, T. Eisenbarth, Simpler, faster, and more robust t-test based leakage detection, 2016
16. S. Sonar, D.B. Roy, R.S. Chakraborty, D. Mukhopadhyay. Side-channel watchdog: run-time evaluation of side-channel vulnerability in FPGA-based crypto-systems. IACR Cryptology ePrint Archive, 2016:182 (2016)
17. Y. Ishai, A. Sahai, D. Wagner, Private circuits: securing hardware against probing attacks, in *In Proceedings of CRYPTO 2003* (Springer, Berlin, 2003). pp. 463–481
18. M. Rivain, E. Prouff, Provably secure higher-order masking of AES. in *12th International Workshop Cryptographic Hardware and Embedded Systems, CHES 2010*, ed. by S. Mangard, F.-X. Standaert , Santa Barbara, CA, Aug 17–20, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6225 (Springer, Berlin, 2010), pp. 413–427
19. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, L. Wingers, The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404 (2013)
20. J. Park, A. Tyagi, *t*-Private logic synthesis on FPGAs. in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 63–68, June 2012
21. D.B. Roy, S. Bhasin, S. Guilley, J. Danger, D. Mukhopadhyay, From theory to practice of private circuit: a cautionary note, in *33rd IEEE International Conference on Computer Design, ICCD 2015*, pp. 296–303. New York City, NY, Oct 18–21, 2015. [21]
22. É. Brier, C. Clavier, F. Olivier, Correlation power analysis with a leakage model, in *Cryptographic Hardware and Embedded Systems*. Lecture Notes in Computer Science, vol. 3156 (Springer, Berlin, 2004), pp. 16–29, Aug 11–13, Cambridge, MA
23. A. Moradi, Oliver Mischke. Glitch-free implementation of masking in modern FPGAs, in *HOST* (IEEE Computer Society, New York, 2012), pp. 89–95, June 2–3 2012. Moscone Center, San Francisco, CA. doi:10.1109/HST.2012.6224326

24. G. Goodwill, B. Jun, J. Jaffe, P. Rohatgi, A testing methodology for side-channel resistance validation. NIST Non-Invasive Attack Testing Workshop, Sept 2011
25. J. Cooper, G. Goodwill, J. Jaffe, G. Kenworthy, P. Rohatgi. Test vector leakage assessment (TVLA) methodology in practice, in *International Cryptographic Module Conference*, Holiday Inn Gaithersburg, MD, Sept 24–26, 2013

Part III
Effective Countermeasures

Chapter 7
Hardware Hardening Approaches Using Camouflaging, Encryption, and Obfuscation

Qiaoyan Yu, Jaya Dofe, Yuejun Zhang, and Jonathan Frey

7.1 Introduction

Under the current global business model, the integrated circuit (IC) supply chain typically involves multiple countries and companies which do not use the same regulation rules. Despite reducing the cost of fabrication, assembly and test, the globalized IC supply chain has led to serious security concerns. Intellectual property (IP) piracy, IC overbuilding, reverse engineering, physical attack, counterfeiting chip, and hardware Trojans are recognized as critical security threats on maintaining the integrity and trust of ICs [6, 7, 14, 24, 27, 29–32, 38, 45, 47, 48, 63]. Those threats could cause companies lose profit and challenge the national security [1]. Hardware counterfeiting and IP piracy cost the US economy more than 200 billion dollars a year on defense and military related chips [33]. Growing hardware IP piracy and reverse engineering attacks challenge traditional chip design and fabrication [24, 29, 32, 47, 48, 50]. Thus, it is imperative to develop countermeasures to address those attacks.

IP piracy is illegal or unlicensed usage of IPs. An attacker can steal valuable hardware IPs in the form of register-transfer-level (RTL) representations ("soft IP"), gate-level designs directly implementable in hardware ("firm IP"), or GDSII design database ("hard IP"), and sell those IPs as genuine ones [10]. Hardware IP

Q. Yu (✉) • J. Dofe
University of New Hampshire, Durham, NH 03824, USA
e-mail: qiaoyan.yu@unh.edu; jhs49@wildcats.unh.edu

Y. Zhang
Ningbo University, Ningbo, Zhejiang, China
e-mail: zhangyuejun@nbu.edu.cn

J. Frey
Charles Stark Draper Laboratory, Inc., Cambridge, MA 02139, USA
e-mail: jfrey@draper.com

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_7

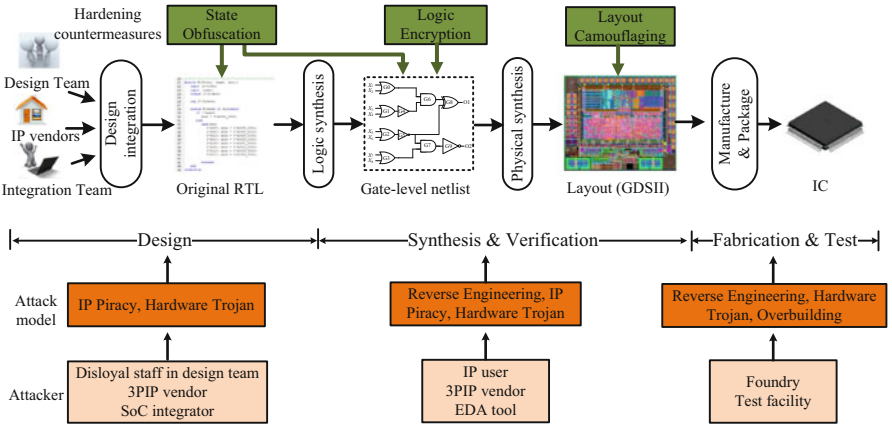


Fig. 7.1 Applying three hardware hardening techniques in the IC design flow

reusing in systems-on-chip (SoCs) design is prevalent practice in silicon industry as it reduces design time and cost dramatically [8]. The IP piracy attack can take place at various stages in the IC supply chain. As shown in Fig. 7.1, the potential IP piracy attackers could be designer, third-party IP (3PIP) vendor, and SoC integrator at design, synthesis, and verification stages. In the fabrication stage, an untrusted foundry may overbuild the IP cores and sell them under a different brand name to make profit.

Reverse engineering of an IC is a process of identifying its structure, design, and functionality [43]. Product teardowns, system-level analysis, process analysis, and circuit extraction are different types of reverse engineering [53]. Using reverse engineering, one can: (1) identify the device technology [29], (2) extract gate-level netlist [53], and (3) infer the chip functionality [24]. Several techniques and tools have been developed to facilitate reverse engineering [15, 37, 53]. Traditionally, reverse engineering is a legal process as per US Semiconductor Chip Protection Act for teaching, analysis, and evaluation of techniques incorporated in mask work [25]. However, reverse engineering is a double-edged sword. One could use reverse engineering techniques to pirate ICs. Reverse engineering attack can be done at various abstraction level of supply chain depending upon the attacker's objectives.

The main focus of this chapter is to review hardware hardening techniques to counteract with IP piracy and reverse engineering attacks. More specifically, we devote this chapter to study active countermeasures like camouflaging, logic encryption, and obfuscation. Different terminology used in this chapter is summarized in Sect. 7.2. The state-of-the-art hardware design hardening methods are reviewed in Sect. 7.3. In Sect. 7.4, a dynamic deflection-based design obfuscation method is presented. Furthermore, we introduce design obfuscation for three-dimensional (3D) ICs in Sect. 7.5. We conclude this chapter in Sect. 7.6.

7.2 Terminology

Before we start to introduce countermeasures for hardening at different levels, we define the terminology that will be used in the remainder of this chapter. Due to similar concepts having different names in the literature published in this field, we categorize the hardware hardening methods into three classes:

(1) *Camouflaging*: Camouflaging is a means of disguising the original presence of the design, more specifically, at layout level. This is a type of countermeasure against the image-analysis-based reverse engineering attack.
(2) *Logic Encryption*: Logic locking, logic obfuscation, and logic encryption are three close concepts that are applied to counteract piracy and reverse engineering attacks of combinational logic circuits. The main principle of logic locking/obfuscation/encryption is to insert key-controlled logic gates to the original logic netlist such that the modified netlist will not function correctly without the right key. Hereafter, we use logic encryption to refer this category. Ideally, the key-controlled logic gates are mixed with the original logic gates; only the countermeasure designer would be able to differentiate the protection gates from the original ones.
(3) *State Obfuscation*: Obfuscation is often utilized to obscure sequential circuits. Similar to logic encryption, a key sequence controls the correct state transitions. State obfuscation typically preserves the original state transitions and adds several key-authentication states before the power-up state. Theoretically, the success rate of IP piracy or reverse engineering attacks is extremely low if the attacker uses brute force methods to find the key.

7.3 State of the Art

This section summarizes three main categories of hardware hardening methods that can resist reverse engineering and IP piracy attacks.

7.3.1 Camouflaging

Circuit camouflaging is a technique that makes subtle changes within the physical layout of a logic cell to obfuscate its actual logic function [18, 19, 43]. The main goal of circuit camouflaging is to disguise the circuit against a reverse engineer who utilizes scanning electron microscopy (SEM) pictures to recover the original chip design. For instance, from the SEM image analysis, a camouflaged logic cell appears as a 2-input NAND gate; however, that logic cell is a 2-input NOR gate in reality. Such a mislead could be achieved by a subtle change on a metal contact. As shown in Fig. 7.2, the contact between metal 1 (M1) and the dielectric is fully connected

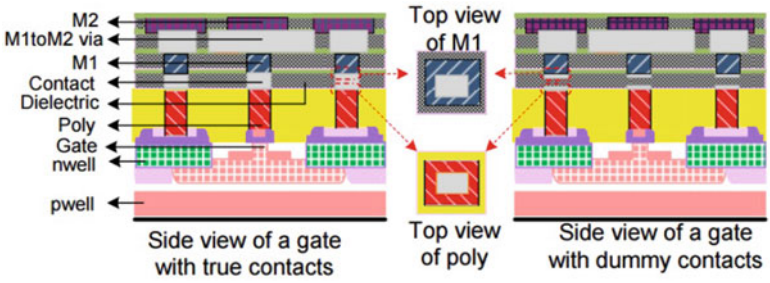


Fig. 7.2 Cross-section views of non-camouflaged contact (*left*) and camouflaged contact (*right*) [43]

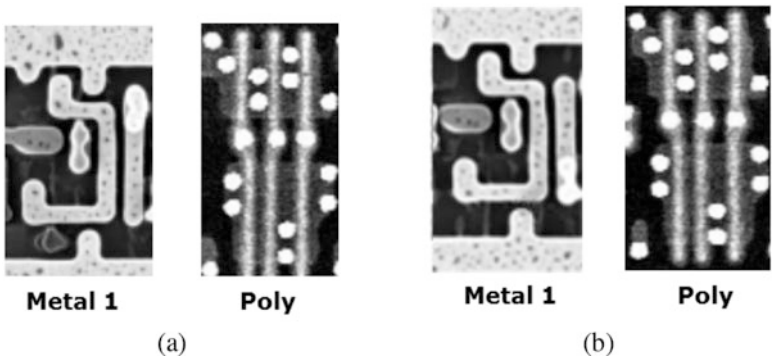


Fig. 7.3 SEM images of (**a**) conventional and (**b**) Camouflaged 2-input AND gate [19]

in the left cell but disconnected after a thin layer of contact material in the right cell. The top views of M1 (and polysilicon) for the gates w/wo camouflaging are identical. The work [19] illustrates that the conventional and camouflaged 2-input AND gates have the same SEM image, as shown in Fig. 7.3. As a result, the attacker could easily extract a wrong netlist if he/she relies on the SEM image analysis. Furthermore, because of the wrong netlist extraction, the attacker cannot precisely modify the netlist to insert hardware Trojans.

The general principle of camouflaging is either to make the connected nodes appear to be isolated, or to have the isolated nodes appear to be connected. In real applications, one can partially camouflage the circuit using a strong custom cell camouflage library [16, 17, 19] or a generic camouflaged cell layout [43]. In the camouflage library [19], every cell looks identical at every mask layer to prevent automated layout recognition [16, 17]. However, making every cell identical in terms of size and spacing with other cells will limit area optimization for the improvement of cell performance. The generic camouflaged cell [43] can perform either as an exclusive-OR (XOR), NAND, or NOR gate, depending on whether true or dummy contacts are used in the cell layout. Rajendran et al. [44] analyze the vulnerability of randomly chosen nodes for camouflaging. Their work shows that

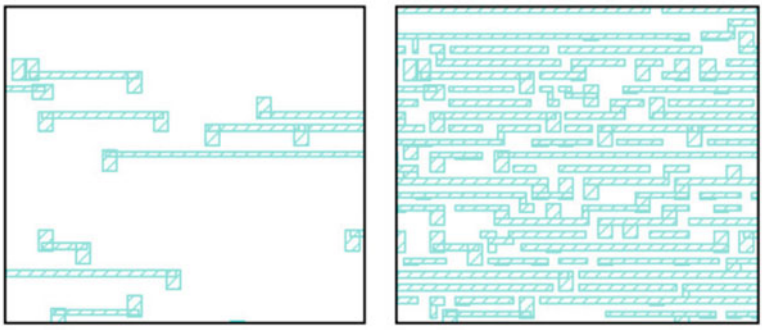


Fig. 7.4 Layout before (*left*) and after (*right*) using camouflaged smart fill of poly, contact, and active layers [19]

the reverse engineering on the camouflaged circuit can still reveal the logic function if the attacker exploits the circuit outputs together with SEM images. Rajendran et al. [43] further propose a smart camouflaging algorithm to increase the difficulty for reverse engineering. The camouflaged standard cells can also be applied to every logic gate [19]. As pointed out above, to camouflage the cells, different logic gates are designed in a way that every cell has the same size and spacing to its neighbor cells. This contradicts the use of transistor sizing for delay optimization. Hence, completely camouflaging the entire chip will sacrifice the system performance, despite the gain of resistance against reverse engineering and piracy attacks. To address this limitation, Cocchi et al. provide elementary block camouflage libraries that are derived from each logic gate layout [17] and could be programmed after the manufacturing process.

In addition to camouflaging, the contacts in a logic cell, metal, active, and device layers can be filled with camouflaged smart fills, which consume all silicon area in the processed layers [19]. Figure 7.4 shows an example of before and after using camouflaged smart fills. As a result, there is no space for hardware Trojan insertion. Meanwhile, the smart fill adds one more task for the attacker to differentiate the real connection trace from the dummy one before he/she identifies the true contacts within one logic cell.

7.3.2 Logic Encryption

Logic encryption methods insert additional logic gates to the original logic netlist, where the additional logic gates are controlled by key bits. An incorrect key leads to netlist mismatch compared to the original netlist and produces different functional outputs than the original ones. The early work [48] introduces the concept of logic encryption as shown in Fig. 7.5. Physical Unclonable Function (PUF) circuit produces a unique challenge. The user key can be stored in a tamper-proof memory.

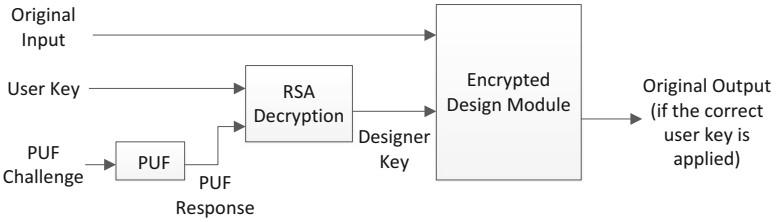


Fig. 7.5 Example of logic encryption structure

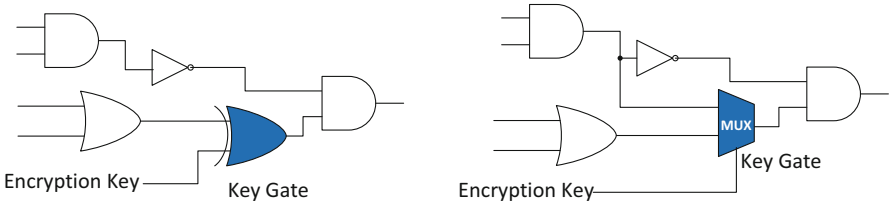


Fig. 7.6 Examples of lightweight logic encryption based on (**a**) XOR and (**b**) MUX key gates. Note, the *solid gates* are key gates and the *unsolid gates* belong to the original netlist

The correct designer key to unlock the encrypted design module is derived from the RSA [49] decryption unit, in which the user key and PUF response serve as ciphertext and crypto key, respectively.

As RSA [49] decryption unit costs 10,000 gates in its ASIC implementation [48], researchers propose to replace RSA using lightweight logic encryption cells, such as XOR/XNOR [40, 51]. Alternatively, multiplexers (MUX) [39, 46, 56] can also be used to encrypt the logic. Figure 7.6 shows examples of simplified logic encryption. In the XOR-based encryption [40, 41, 46, 48, 51, 60], an incorrect key bit may flip the primary output. In the MUX-based encryption [39, 46, 56], an arbitrary internal net is added as an input for the key gate. Without the correct key, the key gate may select the arbitrary internal net for the primary output computation.

7.3.2.1 Attacks Against Logic Encryption

Although combinational logic can be hardened by logic encryption methods mentioned above, there exist several specific attacks that could defeat this type of countermeasures by reducing the number of brute force attempts for key recovery. The existing attack models assume that the attacker owns a locked logic netlist, and he/she either has a copy of the activated chip (as a black box) from the open market or is able to measure the correct chip outputs with given test patterns.

The work [41] demonstrates a sensitization attack, which generates key-leaking input patterns and then passes the key bits to the primary outputs with those input patterns. Lee and Touba [35] propose to perform the brute force attack on the logic cone with the fewest key bits; after the subset of a key vector is retrieved, they continue the same attack iteratively on the other logic cones until all key bits are found. Subramanyan et al. propose a stronger satisfiability checking (SAT) attack [52] that can defeat the encrypted circuit against fault-analysis attacks [41, 46, 61]. The SAT attack is based on the observation that the encrypted netlist with the correct key should generate the same outputs as those from the activated chip, no matter what input pattern is applied to the primary input. Instead of searching for the correct key, the SAT attack looks for a member of the equivalence class of keys that generate the correct output for all input patterns. The SAT attack iteratively searches for the distinguishing inputs (DIP) that cause the inconsistency between the outputs from the netlist using a wrong key and from the unlocked netlist. When no more DIP can be found, the equivalent class of keys is reached. In a hill-climbing attack [39], a key bit is randomly flipped to check whether the output of the netlist with the new key matches to that of the unlocked netlist. The output inconsistency reveals the correct key bit by bit.

7.3.2.2 Logic Encryption Algorithms

The original proposal for logic encryption is EPIC [48], in which XOR/XNOR key gates are randomly placed in the IC. Fault-analysis techniques and tools [34] are exploited to perform stronger logic encryption [41, 46], through which the Hamming distance between the outputs after applying the valid key and an incorrect key is maximized close to 50 %. AND or OR gates can also be used to encrypt the design [23]. In an XOR/XNOR gate locking scheme, a key gate is either an inverter or a buffer. An incorrect key turns the key gate into an inverter. The key gate with the correct key acts as a buffer or a delay unit. Instead of inserting XOR/XNOR key gates, Baumgarten et al. [5] suggest dividing the entire IP into fixed and configurable portions. The configurable portion is implemented as a key-controlled lookup table (LUT). An adversarial foundry is able to observe the geometry of the fixed circuit portion but does not have a clear picture of the configurable portion. This is because the geometry of the configurable circuit portion is composed of generic memory layout, which is programmed by the IP creator after chip fabrication. Encryption using configurable logic is also discussed in the literature [36], in which the logic type can be changed after fabrication.

SAT attacks are capable of breaking most of encrypted logic within a few hours, with the assistance of state-of-the-art SAT solvers. An AES module is utilized to protect the key inputs and thus increase the time for solving a SAT formula [61]. To reduce the hardware cost of AES, a lightweight Anti-SAT unit that outputs 0 (1) for an invalid (valid) key is proposed in [58] to mitigate the SAT attack on logic encryption. Alternatively, key scrambling and logic masking techniques are integrated with the early logic encryption work [41] in their recent work [62].

7.3.3 State Obfuscation

State obfuscation is a countermeasure for sequential circuit protection. The main purpose of the obfuscation is to prevent the adversary from obtaining the clear state transition graph. A remote activation scheme [2] is one of the early works to address IP piracy through the concept of obfuscation. The continuous state transition is granted by a key uniquely generated by a PUF. This method triples one of the existing states; only one of the tripled states protected with the correct key will further drive the state transitions. Hereafter, we refer this method as *DUP*. The piracy-aware IC design flow in [48] enriches the RTL description with an on-chip true random number generator and public-key cryptographic engine. The authors extended their work further in [32], in which a channel from the normal state transition to the black hole states is added to prevent the state from returning to the normal operation.

An SoC design methodology hardware protection through obfuscation of netlist (HARPOON) is presented for IP protection [10]. The main principle of this method is to add a key-controlled obfuscation mode before the finite state machine (FSM) enters the normal mode. Theoretically, an adversary cannot reach the normal mode without a correct key sequence. This method does not protect the normal mode states based on the assumption that the key exploration space is too big for the attacker to reach the true power-up state. Their obfuscation concept is extended to the register transfer level in [11]. The RTL hardware description is converted into control and data flow graphs and then dummy states are added to obfuscate the design with an enabling key sequence. In the gate-level obfuscation work [9], the output of the FSM in the obfuscation mode also flips a few selected logic nodes. Their follow-up work [13] increases the number of unreachable states for the obfuscation mode to thwart hardware Trojan insertion.

The work [20] suggests using obfuscation to thwart hardware tampering. Different than using an external key sequence in [10], this work locks the design with an internally generated Code-Word (similar to a key sequence), which is based on a specific state transition path. This method aims at the key-leaking issue in [10, 11, 13]. Although an incorrect key will cause the FSM temporarily deviate from the correct transition path, the normal state transition will resume after one or several state transitions.

The previous key-based obfuscation works [2, 9, 11, 12, 41] provide a strong defense line against the reverse engineering attacks on the fabricated chip without a correct key. This is true because the number of possible key combinations is too large for brute force attack to hit the right key. As fabless IP cores are prevalent in SoC integration, IP vendors should be aware of the advanced attackers, who may use electronic design automation (EDA) tools to analyze the netlist and obtain information like code coverage, net toggle activities, and thermal activities to accelerate the process of revere engineering. Thus, it is imperative to re-evaluate the previous obfuscation methods under the assumption that EDA tools may reveal obfuscation details.

7.4 Dynamic State-Deflection-Based Obfuscation Method

In this section, we present a new state obfuscation method [21], which considers the reverse engineering attack through EDA tools.

7.4.1 Overview of DSD Obfuscation

Our dynamic state-deflection (DSD) method in [21] protects each normal state ST_x with a black hole state cluster B_x, as shown in Fig. 7.7a. If an incorrect key *!KS* is applied to the FSM, a normal state is deflected to its own black hole cluster. The FSM never returns to the normal state once it enters the black hole cluster. Each black hole cluster is composed of multiple states (only three are shown in Fig. 7.7a), rather than a single one. This is because we map each black hole state to a unique incorrect key (to save the hardware cost, multiple incorrect keys can share one black hole state). As we cannot predict which incorrect keys will be applied by the attacker, we propose a *Mapfunc* function to dynamically assign one black hole state to an incorrect key. The state within the black hole cluster does not remain unchanged; instead, it constantly switches to other black hole states. More details of black hole state creation and dynamic transition are provided in Sects. 7.4.2 and 7.4.3, respectively.

Considering that the gate-level netlist does not provide a clear picture of used and unused states, we use a state flip vector *FlipBit* to selectively invert the state bits using the circuit shown in Fig. 7.7b. Thanks to the inherent feedback loops, the FSM state bits will naturally switch over time without the predetermined transition specification. The modification on the state transition graph can be described with a Markov Chain transition matrix, which indicates the probability for a state transition to another state. Let's use $S_0, \ldots, S_{i-1}$ to represent the states for authentication/obfuscation/black hole cluster and use $S_i, \ldots, S_n$ to represent the normal operation states. The Markov Chain matrix of our obfuscation method is shown in Fig. 7.7c. The non-zero $g_{i(n-1)}$ stands for the single transition path from the obfuscation mode to the normal operation mode. The non-zero sub-matrix H (composed of h_{ij}) in the matrix means that our method creates the channel for each normal state to enter one of the black hole states.

The transition matrices for other methods are shown in Fig. 7.8. According to Fig. 7.8a, the DUP method [2] significantly increases one of the self-transition probability (e.g., p_{ii}) and reduces other probabilities on the same row; no transition paths are strictly blocked by the key sequence. If the attacker runs sufficient amount of simulations, he/she will observe a p_{ii} of high probability. Theoretically, overwriting the state register when the state remains at the same value too long could be an effective tampering technique. The two zero sub-matrices for the HARPOON [10] transition matrix shown in Fig. 7.8b obstruct the transition between the obfuscation states and the true functional states except the reset state in the

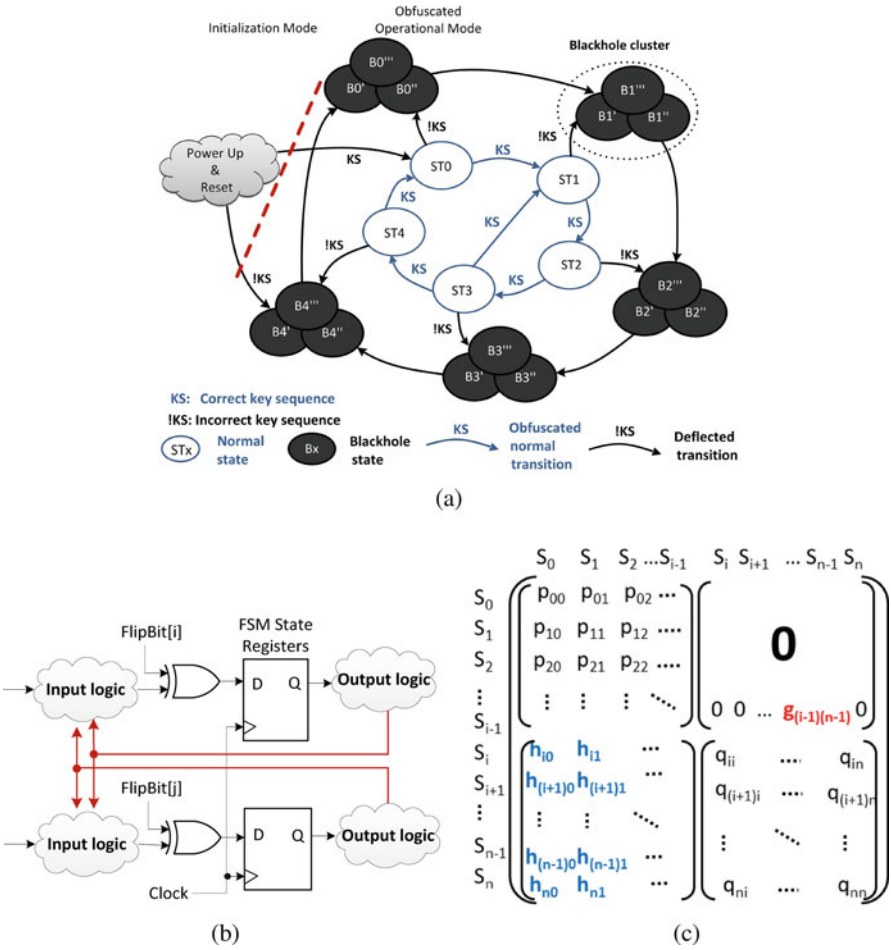


Fig. 7.7 Proposed dynamic state-deflection method for netlist obfuscation. (**a**) State transition graph, (**b**) circuit example of our method, and (**c**) proposed state transition paths represented with a Markov Chain matrix

functional mode (therefore $q_{(i-1)i}$ is the single non-zero probability). The P (element p_{ij}) and Q (element q_{ij}) matrices are strictly confined in its own sub-matrix. If the attacker overwrites the original state, the key obfuscation step can be bypassed. The matrix shown in Fig. 7.8c is the Code-Word-based obfuscation method [20], in which the obfuscated states are multiple ad-hoc states, thus allowing the FSM return to the normal state later.

As can be seen, our method is an extension version of HARPOON and DUP. The main difference is that our H matrix is not a zero matrix, which means the normal states will transition to obfuscation states if an incorrect key is used or if tampering is detected. Intuitively, our method can protect the FSM even if the attacker makes

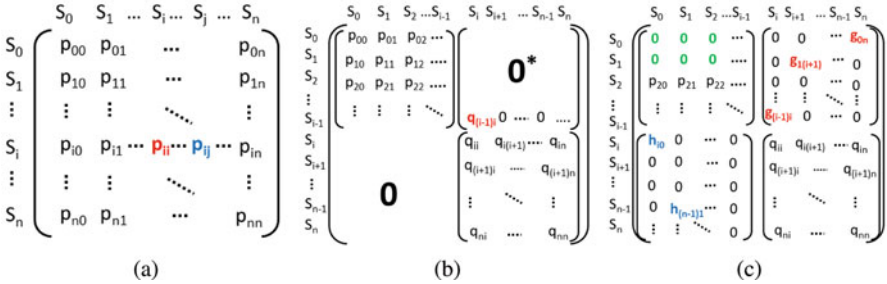


Fig. 7.8 Transition Markov chain matrices. (**a**) DUP [2], (**b**) HARPOON [10], and (**c**) CodeWord [20]

the FSM start in one of the normal states. This is because the FSM could leave the normal mode due to the key authentication before the next state transition. As a result, the workload of reverse engineering dramatically increases than any of the other methods discussed above.

7.4.2 Black Hole State Creation for Gate-level Obfuscation

In the RTL obfuscation methods, the team that adds the dummy states to the obfuscation mode (i.e., the obfuscation mechanism provider) needs to clearly know (1) what states have been used in the normal operation mode, and (2) the exact signals that drive the state transitions. Thus, the obfuscation mechanism provider is assumed to have sufficient knowledge of the original design. The goal of our method is to eliminate this assumption. We propose to obfuscate the gate-level netlist without the prior knowledge of the states in use and signals triggering the state transitions. We believe that the gate-level obfuscation is more attractive and flexible than the RTL obfuscation, as we can decouple the design obfuscation process from the original design.

In our method, we assume that the obfuscation provider is able to learn the number of registers that represent the state binary bits by reading the gate-level netlist. This is reasonable as one can differentiate flip-flop devices from combinational logic gates in the netlist. We form the black hole states shown in Fig. 7.7a with the following procedure [21]. First, each state (e.g., $S3, S2, S1, S0, B3, B2, B1, B0$) in the new FSM is composed of all the original state bits ($S3, S2, S1, S0$) and several newly added flip-flop bits ($B3, B2, B1, B0$). When the correct key sequence is applied, the newly added flip-flops ($B3, B2, B1, B0$) are set to zero and thus the new state keeps consistent with the FSM state before our state extension. This arrangement guarantees that the newly added flip-flops do not change the original function if the key is correct. If $B3, B2, B1, B0$ is non-zero due to an invalid key, then certainly the new state ($S3, S2, S1, S0, B3, B2, B1, B0$) belongs to one of the black hole states in

the cluster. The correct key can be hard-wired in the netlist or saved in a tamper-proof memory. The former one is less secure (but more hardware efficient) than the latter one at a certain degree. Note, more flip-flops added for new states not only increase the difficulty of hardware tampering but also incur a greater hardware cost.

7.4.3 Dynamic Transition in Black Hole Cluster

To raise the difficulty of reverse engineering, we further harden the FSM by creating dynamic transitions within the black hole state clusters. Different than the existing work [2, 10, 13], the transition among the black hole states is not static and predefined. One example of our state transition is shown in Fig. 7.9. If a wrong key sequence (KS) is used, the new state bits are preset to a non-zero vector (e.g., 4'b0001) and the original state bits remain whatever they are defined to be in the original design. After cycle 0, the newly added state bits are modified by a *Rotatefunc* algorithm, which changes the vector $B3, B2, B1, B0$ to a new non-zero vector. Each wrong key sequence will have a unique pattern defined in the *Rotatefunc* algorithm. In the example shown in Fig. 7.9, the *Rotatefunc* is a circular left-shift function. In parallel to the process of switching the black hole state bits, we propose a *Mapfunc* algorithm to deflect the original state bits from the correct state transition. Our *Mapfunc* algorithm takes the newly added state bits as an input to determine how to modify the content of the original state bits. For instance, when $B3, B2, B1, B0$ is 4'b0001, the $S3$ original state bit is flipped. Note, our method selectively inverts the state bit, rather than set the original state bits to a fixed vector. Therefore, the black hole state is not determined in the design time. It depends on which wrong key is used and the corresponding mapping statement specified in the *Mapfunc* algorithm. As shown in Fig. 7.9, the vector of 4'b0010 in the newly added state bits lead to flip $S2$ and $S0$ bits in the original state.

7.4.4 Experimental Results

7.4.4.1 Experimental Setup

To assess the capability against reverse engineering attacks, we applied three representable obfuscation approaches—DUP [2], ISO [12], and HARPOON [10] and our method to the generic gate-level netlist of ISCAS'89 benchmark circuits (http://www.pld.ttu.ee/~maksim/benchmarks/iscas89/verilog/). The obfuscated Verilog codes were synthesized in Synopsys Design Vision with a TSMC 65 nm CMOS technology. The test benches for the synthesized netlists were generated by TetraMax. Verilog simulations were conducted in Cadence NClaunch and the code coverage analyses were performed with the Cadence ICCR tool.

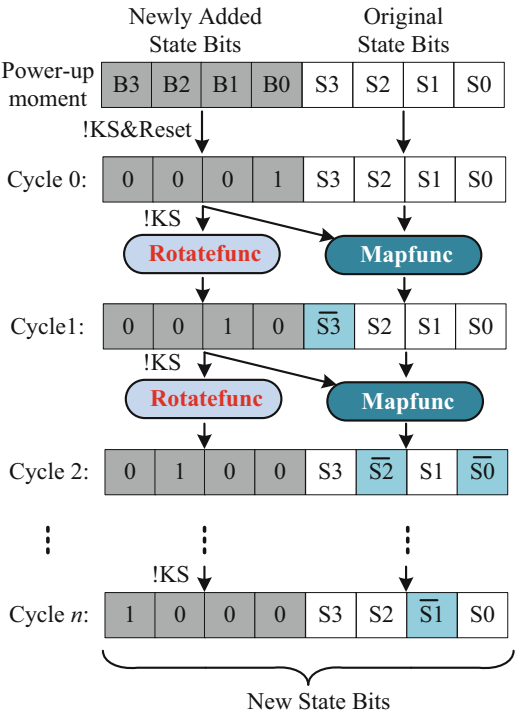


Fig. 7.9 Flow of state transition modification

7.4.4.2 Hardening Capability Against Circuit Switching Activity Analysis Attack

The goal of state obfuscation is to thwart the reverse engineer from retrieving the circuitry used for the normal operation mode. In the existing work, the researchers have not extensively discussed the hardening efficiency of different methods in the scenario that an attacker could exploit the EDA tool to recover the original circuit. In this work, we fill in this gap by studying the net toggle activities and code coverage of the circuits hardened with different obfuscation methods.

Complementary Net Toggle Activities

In the existing methods, the use of an invalid key will either cause the FSM to be stuck in an obfuscation state or an inner loop in a specified obfuscation mode. Thus, the majority of the true circuit under protection will not switch. If the attacker compares the net toggle activities for the cases of using different keys, he/she can sort out the un-toggled nets and recognize those nets are probably used for the true operational circuit. Therefore, one would design the obfuscation method in a way that does not lead to complementary net toggle activities between the scenarios of using the correct key and the wrong keys.

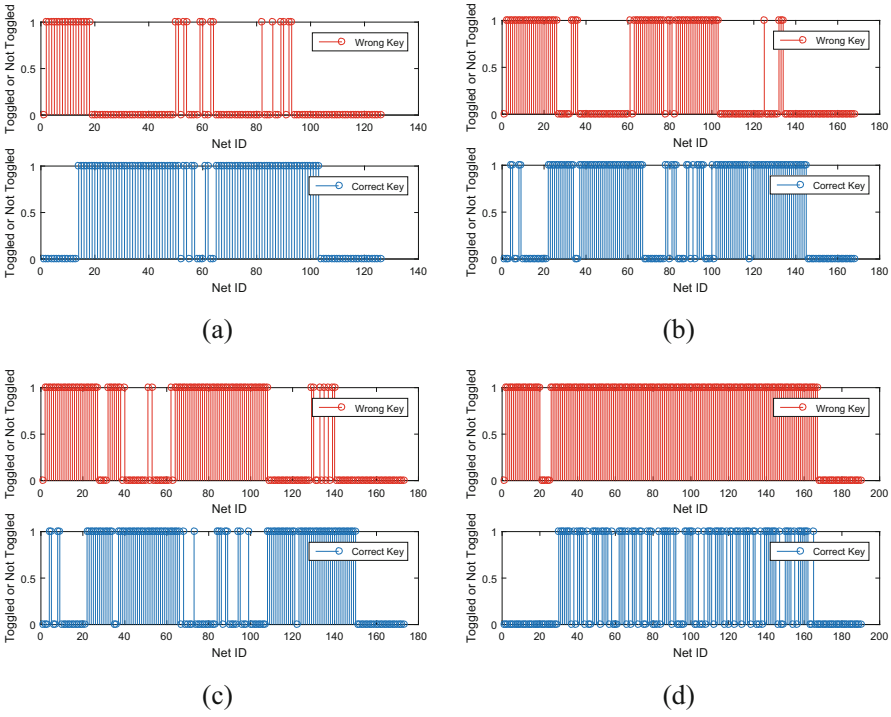


Fig. 7.10 Net toggle activities for s298 obfuscated with (**a**) DUP, (**b**) ISO, (**c**) HARPOON, and (**d**) proposed methods

Figure 7.10a shows the net toggle activities for the s298 circuit protected with the DUP obfuscation method. As can be seen, if incorrect key sequences are applied, the majority of nets from the DUP netlist do not toggle due to the blocked state transitions. In contrast, those un-toggled nets indeed switch if the correct key sequence is used. Thus, the net toggle activities for the case with/without the correct key sequence are nearly complementary. As shown in Fig. 7.10b, c, the number of toggled nets from the ISO and HARPOON netlist is more than that of the DUP netlist, due to the transitions in the obfuscation mode. However, a large number of nets have complementary toggle activities for the obfuscated netlist with a correct key sequence. The proposed obfuscation method examines the key sequence at every state transition and deflects the state to a black hole state. In our method, the FSM does not remain at a small inner loop when an incorrect key is applied to the FSM. Instead, our FSM further changes over time within the black hole state cluster, and triggers the other state registers and output ports to start switching. Consequently, our method obtains nearly the same net toggle activities for the case of with/without the correct key, as shown in Fig. 7.10d. Note, as the test bench generated by TetraMax does not reach 100 % test coverage, some logic outputs in the design module are not toggled for both incorrect and correct key cases.

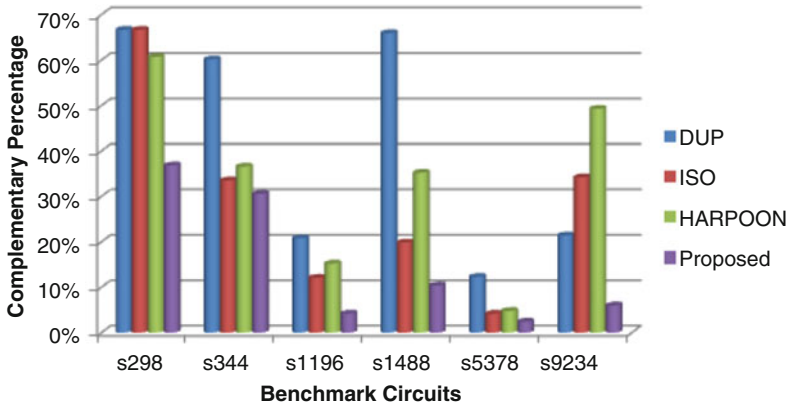


Fig. 7.11 Complementary percentage of the net toggle activities in incorrect and correct key scenarios for different benchmark circuits

To compare the obfuscation efficiency achieved by different methods, we define our comparison metric in Eq. (7.1). This expresses the *complementary degree* of the net toggle activities for the correct and incorrect keys.

$$P_{\text{complement}} = \frac{N_{\text{different toggled nets}}}{N_{\text{total nets in netlist}}} \tag{7.1}$$

A lower complementary degree means that less information will be obtained through code analysis. As shown in Fig. 7.11, the proposed method yields the lowest complementary percentage over the other methods. Based on the experiments performed on the given benchmark circuits, our method reduces the complementary percentage by up to 77 %, 85 %, and 90 % than the DUP, ISO, and HARPOON methods, respectively.

Code Coverage

We used *code coverage* as another metric to compare the hardening degree achieved by different obfuscation methods. If the obfuscated netlist with incorrect key sequences obtains a higher code coverage, this equivalently indicates that obfuscation method will yield a lower net toggling complementary percentage. Our method tightly blends the dummy states with the original states, so that the attacker cannot determine the dummy states by using code coverage analysis through tools like Cadence ICCR. As the netlist under attack is a gate-level netlist, only toggle coverage can be measured. Branch and FSM coverage are not eligible in the code coverage analysis tool. In the method [12], an incorrect key sequence will lead to the FSM enter an isolated mode eventually, thus, the FSM will freeze and the code coverage will be limited. The same reasoning applies to the method [10]. The use of black hole states will lock the chip operation. But, if the attacker has the soft IP or firm IP running in a code analysis tool, the un-toggled statements will

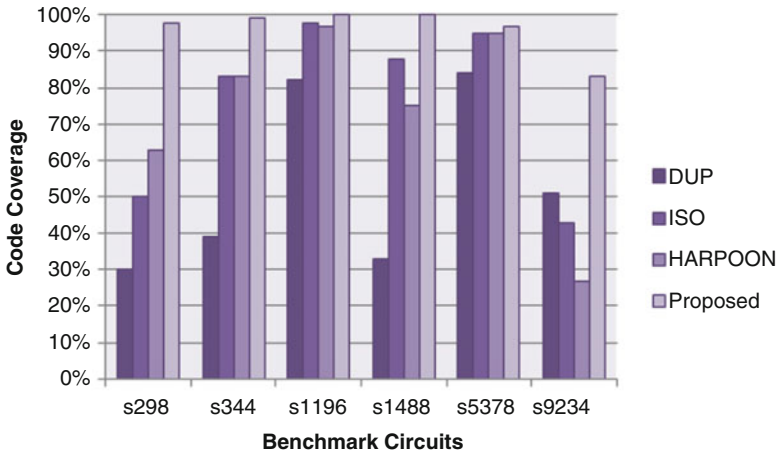


Fig. 7.12 Code coverages for the incorrect key scenarios

reveal the information relevant to the black hole states. As shown in Fig. 7.12, the proposed method always achieves the highest code coverage among the methods under comparison. As our method keeps the state transiting all the time, our method improves the code coverage by up to 56 % over the existing methods if a wrong key sequence is applied. Other methods, either stuck at the wrong states, or enter a small internal obfuscation states, thus having a lower code coverage. Code coverage proves that the proposed method reveals less information with an incorrect key over the other obfuscation methods.

7.4.4.3 Number of Unique State Register Patterns in Obfuscation Mode

In the previous work [10, 11, 13], the state registers either remain same or switch within a loop of different patterns if an incorrect key sequence is applied. Both of the scenarios will facilitate the attacker to identify the states in the obfuscation mode. Then, the attacker can add new logic to overwrite the state registers and thus skip the livelock of the obfuscation state transition. In contrast, our method utilizes *Mapfunc* function to "pseudo-randomly" change the state register values. Even though an attacker may know a *Mapfunc* function is applied in the obfuscation mode, the fact that they do not have knowledge of the exact *Mapfunc* being used will make it difficult to perform an attack where a register is overwritten to a specific value. As shown in Fig. 7.13, our method generates the largest number of unique state patterns over the other methods. On average, the proposed method produces 81, 75, and 75 more unique obfuscation state patterns over DUP, ISO, and HARPOON, respectively. This observation confirms that our method does not only obfuscate the state transitions, but is also capable of resisting the register pattern analysis and thus thwart state register overwrite attacks.

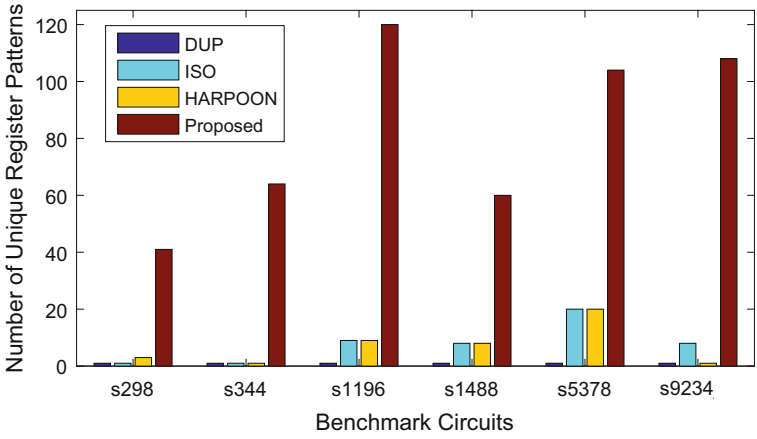


Fig. 7.13 Number of unique register patterns in different benchmark circuits

Table 7.1 Area cost in a 65 nm technology (unit: μm^2)

Benchmark	DUP	ISO	HARPOON	Proposed
s298	283.32	392.04	403.92	345.96
s344	316.08	432.72	445.32	386.64
s1196	737.28	861.12	873.72	825.12
s1488	657.72	826.56	839.16	779.36
s5378	2599.55	3282.48	3298.68	2892.6
s9234	3080.16	2826.72	2840.76	2395.8

7.4.4.4 Area and Power Overhead

The silicon area cost of the benchmark circuits with different obfuscation methods are compared in Table 7.1. The same key size of 12-bits is applied to all the methods. Due to the use of simple logic for state-deflection and dynamic black hole state transitioning, our method reduces the area overhead by up to 12 % over ISO and HARPOON. Compared to DUP, our method consumes more area but provides better IP hardening performance as discussed in Sects. 7.4.4.2 and 7.4.4.3.

The power consumption is compared in Table 7.2. We set up the clock period as 1ns for all designs. As our method checks the key sequence and deflect the FSM state at every state transition cycle, it, on average, consumes 9.5 % and 7.8 % more power than ISO and HARPOON, respectively. Again, the power of DUP is 19 % less than our method, but the DUP method loses obfuscation strength over our approach.

Table 7.2 Total power consumption in a 65 nm technology (unit: mW)

Benchmark	DUP	ISO	HARPOON	Proposed
s298	0.1135	0.1565	0.1597	0.1851
s344	0.1268	0.1704	0.1748	0.1865
s1196	0.1883	0.2341	0.2374	0.2299
s1488	0.063	0.1116	0.1140	0.1139
s5378	1.6886	1.4693	1.4736	1.7650
s9234	1.6713	1.2287	1.2384	1.3355

7.4.5 Discussion

Key-based design obfuscation methods have become attractive to resist reverse engineering and IP piracy attacks. The existing RTL obfuscation methods require the prior knowledge of all the used and unused states and lack protection on the normal operation states. To address the aforementioned limitations, we propose a dynamic state-deflection method for gate-level netlist. Our analyses on the code coverage and net toggle activities show that the proposed method reveals the least amount of information for the adversary who has access to the obfuscated netlist and conducts state register overwrite attacks. Simulation results show that our method achieves up to 56 % higher code coverage than the existing methods if an incorrect key sequences are applied. Hence our method provides a better hardening solution against the adversary who utilizes brute force attacks and EDA code analysis functions to recover the original design. Due to the dynamic deflection feature, on average, our method generates 75 more unique state register patterns than the HARPOON method if an incorrect key is applied, at the cost of 7.8 % power increase in the case study.

7.5 Obfuscation for Three-Dimensional ICs

In this section, we review how three-dimensional (3D) structure can be leveraged to address the security threats in 2D ICs and potential risks of using 3D ICs. Next, we introduce the method [22] that extends the obfuscation concept to 3D ICs.

7.5.1 Leveraging 3D for Security

3D integration has attracted a significant amount of attention during the past two decades to develop diverse computing platforms such as high performance processors, low power systems-on-chip (SoCs), and reconfigurable platforms such as FPGAs. The majority of the existing work has leveraged the unique 3D characteristics to enhance the security of 2D ICs rather than investigating hardware

security within stand-alone 3D ICs [4]. As indicated by leading 3D manufacturing foundries [3], the stacking process of 3D ICs conceals the details of the circuit design and therefore thwarts reverse engineering attacks. Secondly, 3D ICs facilitate *split manufacturing* where the entire IC is distributed throughout multiple dies/planes. Thus, due to the incompleteness of each plane, the design details of a functional block are not revealed.

Existing 3D split manufacturing approaches fall into two primary categories. In the first category, as investigated by Valamehr et al. [54, 55], the entire design is separated into two tiers: one plane is dedicated to being the primary computation plane whereas the second plane is an optional control plane that should be provided by a trusted foundry. This control plane is used to monitor possible malicious behavior within the computation plane and overwrites the malicious signals, if necessary. The second category, as studied by Imeson et al. [28], relies on the interconnects of a trusted die to obfuscate the entire 3D circuit. Thus, the circuit within the untrusted tier cannot be reverse engineered since interconnectivity is unknown. Similar studies have been performed to further enhance the obfuscation level achieved by split manufacturing [42, 57, 59]. As exemplified by these studies, existing approaches rely primarily on the presence of a trusted plane. While effective to enhance security, these existing techniques do not investigate the potential security weaknesses inherent to 3D ICs.

7.5.2 Trustworthiness of Vertical Communication

A non-trivial security weakness within 3D ICs is the trustworthiness of vertical communication. In a heterogeneous 3D stack with dies from different vendors, one of the dies can attempt to extract secret information such as an encryption key, authentication code, or certain IP characteristics. Thus, a die within a 3D stack should not only be protected from external attacks (as is the case in traditional 2D ICs), but also from attacks originating from a nearby die within the same 3D stack. Furthermore, since the authentication level of each die is different, the security of the overall 3D IC is dependent upon the weakest die. The overall 3D IC (including the die with a strong authentication level) can be compromised once an attacker succeeds in accessing the weak die. Note that due to high bandwidth inter-die vertical communication, an attacker has more access points to compromise security, producing additional security threats that do not exist in 2D ICs.

Another potential weakness is the leakage of connectivity information from an untrusted 3D integration foundry. Existing approaches that rely on split manufacturing typically assume that existing vertical communication is inherently secure. Unfortunately, this assumption is not always true. For example, the foundry that manufactures vertical interconnects may leak this connectivity information, resulting in weaker design obfuscation than what is assumed with split manufacturing.

7.5.3 *Proposed Obfuscation Method for 3D ICs*

We propose a new countermeasure for 3D ICs in [22], which is fundamentally different from the split manufacturing approach. In the existing split manufacturing methods, one of the dies and vertical connections must be manufactured by a trusted foundry. Without this assumption, split manufacturing cannot ensure the trustworthiness of a 3D SoC with dies from different vendors. It is predicted that commercial dies, rather than customized dies, will be vertically integrated to develop 3D ICs in the near future. Since the I/O definition and certain specifications of commercial dies are public, an attacker can reverse engineer the 3D SoC design, particularly if each die of the 3D IC can be separated from the stack using a debonding technique.

7.5.3.1 Overview of Proposed 3D Obfuscation Method

To eliminate the need for at least one trusted foundry for 3D ICs, we propose a secure cross-plane communication network [22]. The proposed method is based on the insertion of a *Network-on-Chip (NoC)-based shielding plane* between two commercial dies to thwart reverse engineering attacks on the vertical dimension. As shown in Fig. 7.14, the proposed NoC shielding plane (NOC-SIP) obfuscates the communication among the adjacent dies. As compared to split manufacturing, this method provides higher flexibility and scalability when developing more secure 3D ICs. This characteristic is due to the enhanced modularity and scalability of NoCs as compared to a bus-centric communication network. Ad-hoc algorithms that split IC functionalities are not suitable for a large-scale system. Furthermore, additional wire lifting through split manufacturing leads to a larger number of Through-Silicon-Vias (TSVs), resulting in a larger area overhead and TSV capacitance (and therefore power).

The essence of the proposed NOC-SIP is to provide an obfuscated communication channel between two planes that host commercial dies. Our method makes it significantly more challenging to reverse engineer the 3D IC system. If the proposed shielding layer is sufficiently strong, the 3D system has more flexibility to use low-end dies without sacrificing the overall system's security assurance. We assume each commercial die has a regular I/O pad map. Those I/O pads are connected to the proposed NOC-SIP with a regular node array, as shown in Fig. 7.14. Therefore, the specific I/O connectivity information of the dies is hidden to the 3D foundry. As a result, the proximity attack [42] is less likely to help to reveal the design details from split dies.

An example of the proposed method is depicted in Fig. 7.15. Without the proposed NOC-SIP die, node A in plane 2 would be connected to node B in plane 1 directly with a TSV. If an attacker has the ability to reverse engineer debonded planes 1 and 2, the 3D IC would be compromised. Alternatively, the NOC-SIP plane redirects the signal from a TSV-A on node A to several routing

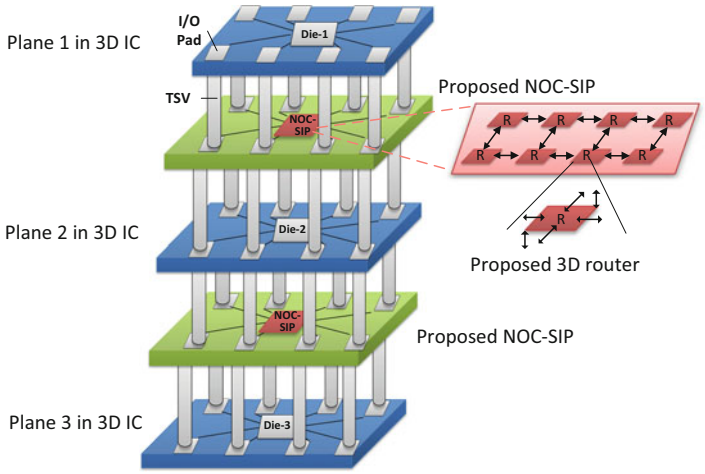


Fig. 7.14 Conceptual representation of the proposed countermeasure for 3D IC security

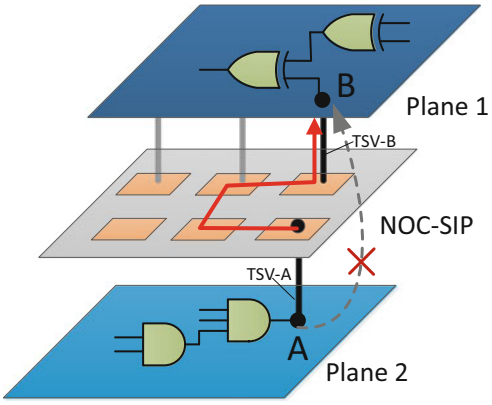


Fig. 7.15 An example of vertical communication through the proposed NOC-SIP

hops before the signal truly reaches TSV-B on node B. Thus, the direct connection from node A to node B is removed. The proposed NOC-SIP enhances security for the following three reasons: (1) even if the adversary successfully separates planes 1 and 2, a scanning electron microscope (SEM) picture of the vertical connection does not reveal useful information for reverse engineers to retrieve the complete system design, (2) the inserted NOC-SIP facilitates the use of 2D security countermeasures to address 3D security threats, and (3) the inherent scalability of the NOC-SIP overcomes the limited flexibility and scalability concerns in the existing split manufacturing algorithms.

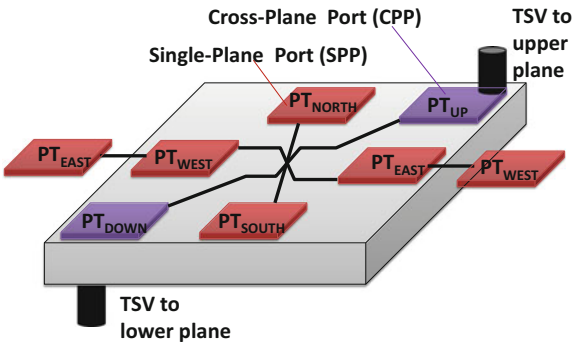


Fig. 7.16 Proposed 3D router architecture

7.5.3.2 Proposed 3D Router Design

The proposed 3D router is shown in Fig. 7.16. The single-plane ports (SPP) PT_{North}, PT_{East}, PT_{South}, and PT_{West} fulfill the communication function within the NOC-SIP. Alternatively, the two cross-plane port (CPP) PT_{UP} and PT_{DOWN} are responsible for cross-plane communication. In contrast, a typical 2D router has five pairs of input and output ports. Figure 7.17a shows the diagram of one input and one output port for a 2D router. The proposed 3D router in the NOC-SIP is different than the router in the traditional 2D NoC. As shown in Fig. 7.17b, the logic of the input and output ports in our 3D router is much simpler than that of a 2D router. Only single-depth input and output FIFOs are required; FIFO controllers are removed; the route computing unit is replaced with a simple routing path request unit (no computation inside); a simpler arbiter can fulfill the routing arbitration. Among the five ports in a 2D router, one port is connected with a network interface (NI) to reach a processing element (e.g., microprocessor or a memory core). However, in the NOC-SIP, no processing element is connected to any port of the 3D router. The packetization function that is executed in the NI of a 2D NoC is part of our vertical router's function.

In our 3D router, the cross-plane ports, PT_{UP} and PT_{DOWN}, are designed for vertical connection from/to other planes. The primary functions of these two ports include: (1) packetize/de-packetize the bit stream from/to the other plane, (2) assign a source-routing path for each data packet, and (3) buffer the bit stream if the previous packets do not have available bandwidth.

As shown in Fig. 7.17c, k-bit data from plane 1 is stored in the buffer before packetization. The packet leaving the cross-plane port is formatted in a way that starts with header information, follows with the detailed routing hops, and ends with the real data. The uniqueness of this cross-plane port is the existence of a router identifier (that is programmable after fabrication) and source routing table, which are provided by the 3D chip designer through one-time read-only memory (ROM) programming equipment. As the router identifier and routing table are initialized after fabrication, placing malicious hardware in the NOC-SIP during the design stage cannot guarantee compromised system. Similarly, the knowledge of the 3D

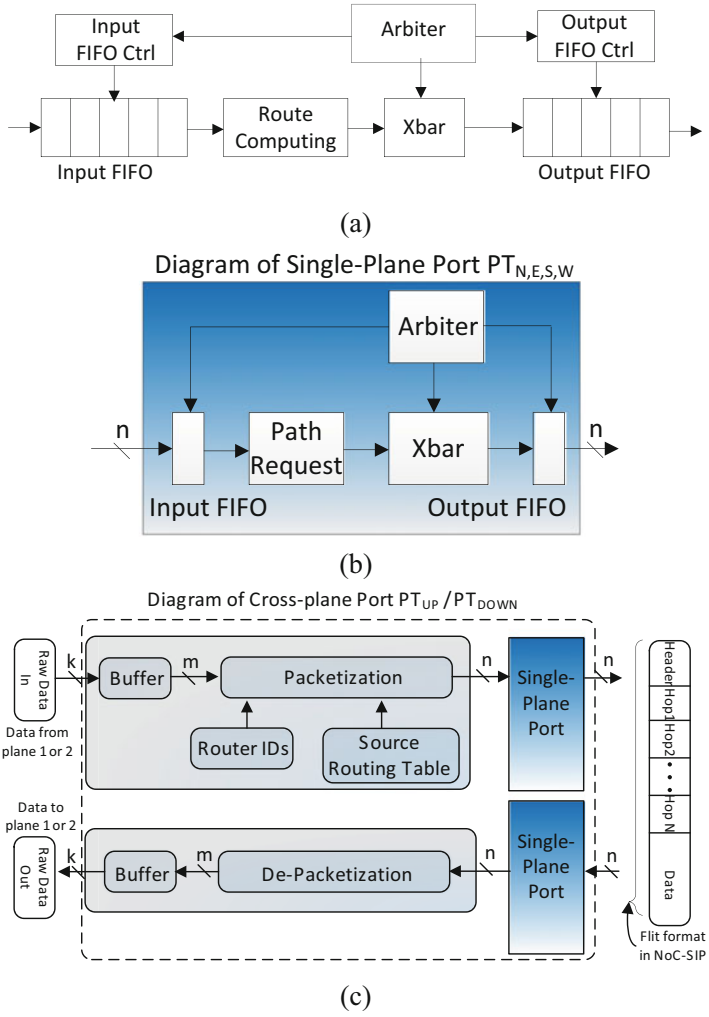


Fig. 7.17 Schematics of (**a**) one pair of input and output ports in a 2D router, (**b**) single-plane port in the proposed 3D router, and (**c**) cross-plane port in the proposed 3D router

router design does not help the reverse engineer, as the router identifier and source routing table are unknown before deployment. Once the router ID or the source routing table is changed, the secret information sniffed by the attack for one case would not be useful.

Source routing introduces unpredictability for the adversary who intends to insert hardware Trojans in the NOC-SIP. As there is no computation unit for route path preparation, the attacker cannot successfully execute a meaningful attack without the knowledge of router identifier assignment. Another advantage of source routing is to manually balance the latency for the transmission between different pairs of

I/O pads on planes 1 and 2, shown in Fig. 7.14. To thwart the delay-based side-channel attacks, the source routing design in the 3D router further facilitates a dynamic routing, which varies the latency of the communication between source and destination ports within the NOC-SIP.

7.5.3.3 Assessment on 3D Obfuscation Method

We implemented the proposed NOC-SIP in Verilog HDL and synthesized the HDL code in Synopsys Design Compiler with a 65 nm TSMC technology. The width of the raw data from a plane (other than NOC-SIP) is set to 5 bits, and the packet width for the NOC-SIP is 32 bits. The input and output FIFOs are 32-bit single-depth buffer. XY routing algorithm is applied to the 2D NoC design. Round-robin arbitration is used in both 2D NoC and NOC-SIP. The NI for the 2D mesh NoC is OCP-IP compatible [26]. The hardware cost of our NOC-SIP is compared with two typical 4×4 mesh NoCs.

The router switching activity of the NOC-SIP is used as a metric to evaluate the difficulty of identifying vertical connectivity through reverse engineering. Three cross-plane communication methods are compared: direct TSV, NOC-SIP with XY routing, and NOC-SIP with source routing. Direct TSV refers to the case where the I/O pads from different dies are statically connected during the 3D IC integration process. NOC-SIP with XY and source routing stand for the proposed shielding layer using XY packet routing and dynamic source routing, respectively.

First, we randomly select a single pair of I/O pads from two planes for vertical communication. The number of switching transitions of each 3D router is recorded in 5000 clock cycles. As shown in Fig. 7.18a, the activity of the TSVs directly connected to two I/O pads indicates that the TSV nodes 4 and 9 are used in the cross-plane communication. Note that the color bar represents the number of node transitions in 5000 cycles. Alternatively, the router activity map (Fig. 7.18b) of the NOC-SIP with XY routing shows that the routers 4, 8, 9, 10, 11, and 12 are used in the cross-plane communication. A reverse engineer, however, cannot know which two TSVs in the routers are actually used for cross-plane communication. The use of source routing further increases the number of involved routers to 10, thereby increasing the degree of obfuscation, as shown in Fig. 7.18c.

If two pairs of vertical communication are applied, the NOC-SIP achieves enhanced obfuscation performance. As shown in Fig. 7.19, the number of active routers in the NOC-SIP is always greater than the direct TSV method. The source routing for NOC-SIP has the ability to distribute the data transmission throughout the entire NOC-SIP. Thus, the proposed approach makes reverse engineering significantly more difficult.

The experiments shown by Figs. 7.18 and 7.19 are extended by increasing the number of communication pad pairs to 10. As shown in Fig. 7.20, the NOC-SIP with source routing engages 12 routers with two pairs of pads. In contrast, the direct TSV method involves 12 routers with 10 pairs of pads. This behavior demonstrates that the proposed NOC-SIP method effectively obfuscates the cross-

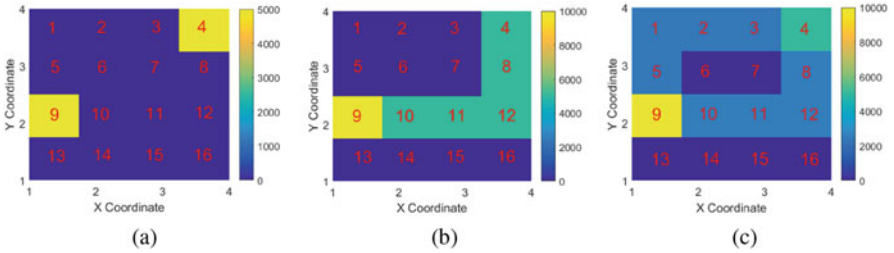


Fig. 7.18 Single pad-to-pad vertical communication. (**a**) Direct TSV, (**b**) NOC-SIP with XY routing, and (**c**) NOC-SIP with source routing

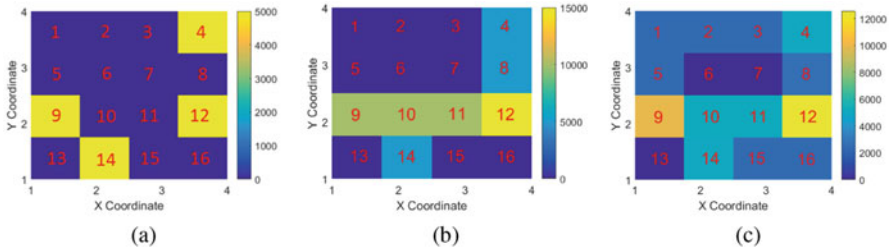


Fig. 7.19 Double pad-to-pad vertical communication. (**a**) Direct TSV, (**b**) NOC-SIP with XY routing, and (**c**) NOC-SIP with source routing

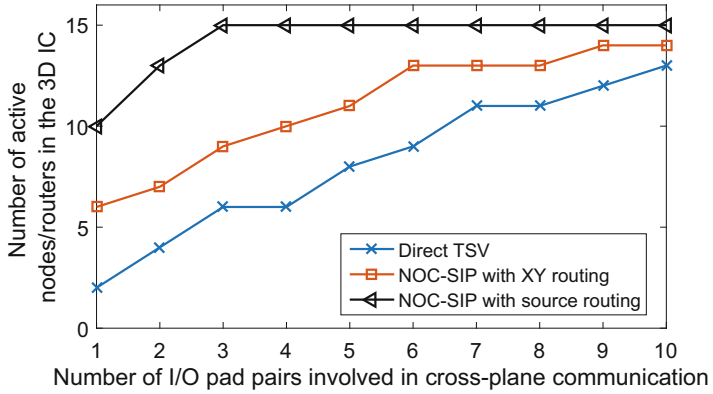


Fig. 7.20 The number of active routers versus the number of routers having incoming data

plane communication by increasing the reverse engineering time as compared to using direct TSV connections. The direct TSV is the most vulnerable method against reverse engineering on vertical connections.

7.5.4 Discussion

Existing works have demonstrated that 3D integration provides additional security defense to thwart hardware attacks in 2D ICs. The security threats that are inherent to true 3D ICs, however, have not received much attention. To thwart reverse engineering attacks on the cross-plane communication (i.e., vertical communication channel), an NOC-SIP layer is proposed. The proposed NOC-SIP obfuscates the vertical communication within a 3D stack. Simulation results demonstrate that the proposed NOC-SIP engages all of the routers as long as more than two pairs of I/O pads from different planes are in use. Alternatively, direct TSV connection method requires ten pairs of I/O pads from two planes to reach the same level of obfuscation as the proposed method. Thus, the proposed NOC-SIP makes it significantly more challenging for a reverse engineer to retrieve the vertical connectivity information.

7.6 Summary

Due to the trend of outsourcing designs to overseas foundries, IC designers and users need to re-evaluate the trust in hardware. The hardware threats such as reverse engineering, IP piracy, chip overproduction, and hardware Trojan insertion are major concerns for hardware security. Key-based obfuscation is one of the promising countermeasures to thwart the hardware attacks like reverse engineering and IP piracy. This chapter summarizes the state-of-the-art hardware hardening methods at layout, gate, and register transfer levels. Camouflaging, logic encryption, and state obfuscation are three representable categories for hardware hardening against malicious reverse engineering and IP piracy attacks. To address the potential security threats in 3D ICs, we also introduce an obfuscation method to protect the vertical communication channels in 3D ICs. As the attackers may obtain extra knowledge and tools to perform reverse engineering and IP piracy attacks, it is imperative to explore new hardware-efficient hardening methods to combat the existing hardware attacks.

References

1. S. Adee, The hunt for the kill switch. IEEE Spectr. **45**, 34–39 (2008)
2. Y. Alkabani, F. Koushanfar, M. Potkonjak, Remote activation of ICs for piracy prevention and digital right management, in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2007), pp. 674–677
3. S. Bansal, 3D IC Design. EETimes (2011). http://www.eetimes.com/document.asp?doc_id=1279081
4. C. Bao, A. Srivastava, 3D Integration: new opportunities in defense against cache-timing side-channel attacks, in *Proceedings of Computer Design (ICCD)* (2015), pp. 273–280

5. A. Baumgarten, A. Tyagi, J. Zambreno, Preventing IC piracy using reconfigurable logic barriers. IEEE Des. Test Comput. **27**(1), 66–75 (2010)
6. D.J. Bernstein, Cache-timing attacks on AES. Technical Report (2005)
7. S. Bhunia, M.S. Hsiao, M. Banga, S. Narasimhan, Hardware trojan attacks: threat analysis and countermeasures. Proc. IEEE **102**(8), 1229–1247 (2014)
8. E. Castillo, U. Meyer-Baese, A. Garcia, L. Parrilla, A. Lloris, IPP@HDL: efficient intellectual property protection scheme for IP cores. IEEE Trans. Very Large Scale Integr. VLSI Syst. **15**(5), 578–591 (2007)
9. R.S. Chakraborty, S. Bhunia, Hardware protection and authentication through netlist level obfuscation, in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2008), pp. 674–677
10. R. Chakraborty, S. Bhunia, HARPOON: an obfuscation-based soc design methodology for hardware protection. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **28**(10), 1493–1502 (2009)
11. R.S. Chakraborty, S. Bhunia, Security through obscurity: an approach for protecting register transfer level hardware IP, in *Proceedings of Hardware Oriented Security and Trust (HOST)* (2009), pp. 96–99
12. R. Chakraborty, S. Bhunia, RTL Hardware IP protection using key-based control and data flow obfuscation, in *Proceedings of International Conference on VLSI Design (VLSID)* (2010), pp. 405–410
13. R.S. Chakraborty, S. Bhunia, Security against hardware trojan attacks using key-based design obfuscation. J. Electron. Test. **27**(6), 767–785 (2011)
14. R.S. Chakraborty, S. Narasimhan, S. Bhunia, Hardware trojan: threats and emerging solutions, in *Proceedings of High Level Design Validation and Test Workshop (HLDVT'09)* (2009), pp. 166–171
15. Chipworks (2012). http://www.chipworks.com/en/technical-competitive-analysis/
16. Circuit camouflage technology (2012). http://www.smi.tv/SMI_SypherMedia_Library_Intro.pdf
17. R. Cocchi, J. Baukus, B. Wang, L. Chow, P. Ouyang, Building block for a secure CMOS logic cell library. US Patent App. 12/786,205 (2010)
18. R. Cocchi, L. Chow, J. Baukus, B. Wang, Method and apparatus for camouflaging a standard cell based integrated circuit with micro circuits and post processing. US Patent 8,510,700 (2013)
19. R. Cocchi, J.P. Baukus, L.W. Chow, B.J. Wang, Circuit camouflage integration for hardware IP protection, in *Proceedings of Design Automation Conference (DAC)* (2014), pp. 1–5
20. A.R. Desai, M.S. Hsiao, C. Wang, L. Nazhandali, S. Hall, Interlocking obfuscation for anti-tamper hardware, in *Proceedings of Cyber Security and Information Intelligence Research Workshop (CSIIRW)* (2013), pp. 8:1–8:4
21. J. Dofe, Y. Zhang, Q. Yu, DSD: a dynamic state-deflection method for gate-level netlist obfuscation, in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2016), pp. 565–570
22. J. Dofe, Q. Yu, H. Wang, E. Salman, Hardware security threats and potential countermeasures in emerging 3D ICS, in *Proceedings of Great Lakes Symposium on VLSI (GLSVLSI)* (ACM, New York, 2016), pp. 69–74
23. S. Dupuis, P.S. Ba, G.D. Natale, M.L. Flottes, B. Rouzeyre, A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans, in *Proceedings of International On-Line Testing Symposium (IOLTS)* (2014), pp. 49–54
24. ExtremeTech, iphone 5 A6 SoC reverse engineered, reveals rare hand-made custom CPU, and tri-core GPU (2012)
25. Federal statutory protection for mask works (1996). http://www.copyright.gov/circs/circ100.pdf
26. J. Frey, Q. Yu, Exploiting state obfuscation to detect hardware trojans in NoC network interfaces, in *Proceedings of Midwest Symposium on Circuits and Systems (MWSCAS)* (2015), pp. 1–4

27. U. Guin, K. Huang, D. DiMase, J.M. Carulli, M. Tehranipoor, Y. Makris, Counterfeit integrated circuits: a rising threat in the global semiconductor supply chain. Proc. IEEE **102**(8), 1207–1228 (2014)
28. F. Imeson, A. Emtenan, S. Garg, M.V. Tripunitara, Securing computer hardware using 3D integrated circuit (ic) technology and split manufacturing for obfuscation. USENIX Security, 13 (2013)
29. Intel's 22-nm tri-gate transistors exposed (2012). http://www.chipworks.com/blog/technologyblog/2012/04/23/intels-22-nm-tri-gate-transistors-exposed
30. P.C. Kocher, Timing attacks on implementations of diffie-Hellman, RSA, DSS, and Other Systems, in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '96 (Springer, London, 1996), pp. 104–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=646761.706156
31. P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in *Advances in Cryptology—CRYPTO'99* (Springer, Berlin, 1999), pp. 388–397
32. F. Koushanfar, Provably secure active IC metering techniques for piracy avoidance and digital rights management. IEEE Trans. Inf. Forensics Secur. **7**(1), 51–63 (2012)
33. L. Frontier Economics Ltd, Estimating the global economic and social impacts of counterfeiting and piracy (2011)
34. H.K. Lee, D.S. Ha, HOPE: an efficient parallel fault simulator for synchronous sequential circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **15**(9), 1048–1058 (1996)
35. Y.W. Lee, N.A. Touba, Improving logic obfuscation via logic cone analysis, in *Proceedings of Latin-American Test Symposium (LATS)* (2015), pp. 1–6
36. B. Liu, B. Wang, Reconfiguration-based VLSI design for security. IEEE J. Emerging Sel. Top. Circuits Syst. **5**(1), 98–108 (2015)
37. T. Meade, S. Zhang, Y. Jin, Netlist reverse engineering for high-level functionality reconstruction, in *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)* (2016), pp. 655–660
38. A. Moradi, M.T.M. Shalmani, M. Salmasizadeh, A generalized method of differential fault attack against AES cryptosystem, *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (Springer, Berlin, Heidelberg, 2006), pp. 91–100
39. S.M. Plaza, I.L. Markov, Solving the third-shift problem in ic piracy with test-aware logic locking. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(6), 961–971 (2015)
40. J. Rajendran, Y. Pino, O. Sinanoglu, R. Karri, Logic encryption: a fault analysis perspective, in *Proceedings of Design, Automation and Test in Europe (DATE)*. EDA Consortium (2012), pp. 953–958
41. J. Rajendran, Y. Pino, O. Sinanoglu, R. Karri, Security analysis of logic obfuscation, in *Proceedings of Design Automation Conference (DAC), ACM/EDAC/IEEE* (2012), pp. 83–89
42. J. Rajendran, O. Sinanoglu, R. Karri, Is manufacturing secure? in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2013), pp. 1259–1264
43. J. Rajendran, M. Sam, O. Sinanoglu, R. Karri, Security analysis of integrated circuit camouflaging, in *Proceedings of ACM SIGSAC Conference on Computer Communications Security*, CCS '13 (ACM, New York, 2013), pp. 709–720
44. J. Rajendran, O. Sinanoglu, R. Karri, VLSI testing based security metric for IC camouflaging, in *Proceedings of IEEE International Test Conference (ITC)* (2013), pp. 1–4
45. J. Rajendran, A. Ali, O. Sinanoglu, R. Karri, Belling the CAD: toward security-centric electronic system design. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(11), 1756–1769 (2015)
46. J. Rajendran, H. Zhang, C. Zhang, G.S. Rose, Y. Pino, O. Sinanoglu, R. Karri, Fault analysis-based logic encryption. IEEE Trans. Comput. **64**(2), 410–424 (2015)
47. M. Rostami, F. Koushanfar, R. Karri, A primer on hardware security: models, methods, and metrics. Proc. IEEE **102**(8), 1283–1295 (2014)
48. J. Roy, F. Koushanfar, I. Markov, EPIC: ending piracy of integrated circuits, in *Proceedings of Design, Automation and Test in Europe (DATE)* (2008), pp. 1069–1074

49. B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd edn. (Wiley, New York, 1995)
50. SEMI, Innovation is at risk as semiconductor equipment and materials industry loses up to $4 billion annually due to ip infringement (2008). www.semi.org/en/Press/P043775/
51. O. Sinanoglu, Y. Pino, J. Rajendran, R. Karri, Systems, processes and computer-accessible medium for providing logic encryption utilizing fault analysis. US Patent App. 13/735,642 (2014)
52. P. Subramanyan, S. Ray, S. Malik, Evaluating the security of logic encryption algorithms, in *Proceedings of Hardware Oriented Security and Trust (HOST)* (2015), pp. 137–143
53. R. Torrance, D. James, The state-of-the-art in IC reverse engineering, in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems* (CHES) (Springer, Berlin, Heidelberg, 2009), pp. 363–381
54. J. Valamehr et al., A qualitative security analysis of a new class of 3-D integrated crypto co-processors. in *Cryptography and Security*, ed. by D. Naccache (Springer, Berlin, Heidelberg, 2012), pp. 364–382
55. J. Valamehr et al., A 3-D split manufacturing approach to trustworthy system development. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(4), 611–615 (2013)
56. J.B. Wendt, M. Potkonjak, Hardware obfuscation using puf-based logic, in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2014), pp. 270–271
57. K. Xiao, D. Forte, M.M. Tehranipoor, Efficient and secure split manufacturing via obfuscated built-in self-authentication, in *Proceedings of Hardware Oriented Security and Trust (HOST)* (2015), pp. 14–19
58. Y. Xie, A. Srivastava, Mitigating sat attack on logic locking. Cryptology ePrint Archive, Report 2016/590 (2016). http://eprint.iacr.org/
59. Y. Xie, C. Bao, A. Srivastava, Security-aware design flow for 2.5 D IC technology, in *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices(TrustED)*. (ACM, New York, 2015), pp. 31–38
60. M. Yasin, O. Sinanoglu, Transforming between logic locking and IC camouflaging, in *Proceedings of International Design Test Symposium (IDT)* (2015), pp. 1–4
61. M. Yasin, J. Rajendranand, O. Sinanoglu, R. Karri, On improving the security of logic locking. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **35**(9), 1411–1424 (2015)
62. M. Yasin, B. Mazumdar, J. Rajendranand, O. Sinanoglu, SARlock: SAT attack resistant logic locking, in *Proceedings of Hardware Oriented Security and Trust (HOST)* (2016), pp. 236–241
63. J. Zhang, H. Yu, Q. Xu, HTOutlier: hardware trojan detection with side-channel signature outlier identification, in *Proceedings of Hardware Oriented Security and Trust (HOST)* (2012), pp. 55–58

Chapter 8
A Novel Mutating Runtime Architecture for Embedding Multiple Countermeasures Against Side-Channel Attacks

Sorin A. Huss and Marc Stöttinger

8.1 Introduction

In today's modern communication technology-based society embedded electronic devices get more and more into our daily life. The purposes of these embedded systems are quite different, but most of them require to transfer data between components within a device or between various devices. At least part of the transferred information is sensitive and thus has to be protected in order to avoid eavesdropping and misuse. In principle this problem may be solved by means of cryptographic algorithms implemented in software. However, the related overhead such as additional power consumption needs to be considered carefully. A more appropriate solution to achieve the required computational performance is to introduce dedicated microelectronic modules to efficiently speed-up these operations. One option is to use FPGA-based ciphers to improve the system performance without losing the important software-like flexibility.

In 1999 new attack methods known as Simple Power Attack (SPA) and Differential Power Analysis (DPA) have been introduced [9], which since then significantly changed the view on design requirements of security sensitive applications. An adversary just needs to observe the physical behavior of the cryptographic module, while it operates in its normal mode and then to exploit the recorded data by analyzing key-dependent properties such as execution time, power consumption,

S.A. Huss (✉)
CASED - Center of Advanced Security Research Darmstadt, and Integrated Circuits and Systems Lab, Computer Science Department, Technische Universität Darmstadt, Hochschulstrasse 10, Darmstadt, Germany
e-mail: huss@iss.tu-darmstadt.de; sorn.huss@cased.de

M. Stöttinger
Independent Researcher (formerly with CASED), Darmstadt, Germany
e-mail: marc.stoettinger@gmail.com

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_8

or electromagnetic radiation. Such passive, non-invasive characteristics make side-channel attacks (SCA) rather dangerous because they are efficient, easy to conduct, and do not leave any evidence of the attempted assault. Especially the power analysis attack is one of the most popular classes of passive SCA strategies and is therefore being thoroughly investigated in academia as well as in industry.

Many countermeasures have meanwhile been proposed aimed to hardening implementations of cryptographic modules against such attacks. The main goal of hardening a design by introducing countermeasures is to reduce and, in the best case, to minimize the amount of exploitable physical information. Thus, next to modifying an encryption algorithm, the envisaged technology platform as well as the architectural structure of the module needs to be adapted accordingly.

Countermeasures against power analysis attacks are mainly applied on the algorithm or on the cell level of the implementation platform. The reason for this fact is that countermeasures are bound directly to the envisaged platform. This means that in case of a software implementation the countermeasures can only be mapped to software as part of the cryptographic algorithm, i.e., on algorithmic level. Masking is a good example for such a countermeasure. In case of exploiting a custom chip as platform, the designer may apply circuit-related countermeasures on cell level such as Dual-Rail logic. Please note that in both cases the physical architecture of the implementation is not being modified. In contrast, the concept of an FPGA platform offers much more flexibility. This is because of the inherent reconfiguration capability of both functional blocks and their interconnection on an FPGA—even at runtime.

In the context of countermeasures, however, only one approach has been published so far which directly addresses the architecture level by exploiting the concept of modifying the physical structure of the platform. The so-called Temporal Jitter method proposed by Mentens et al. in [14] exploits the reconfiguration capability of the data path in order to freely position both registers and combinatoric logic in the circuit. In doing so, the activity pattern of each clock cycle is being modified and, similar to Shuffling on algorithmic level, the adversary cannot directly deduce from the recorded power traces which specific operation is being processed. Platform reconfiguration is therefore the key method to accommodate on all abstraction levels highly variable, at runtime mutating architectures as addressed in this chapter.

The paper is structured as follows. Section 8.2 introduces the basic concepts, multiple countermeasures resulting from a tailored digital system design methodology, and the advocated architecture. Section 8.3 summarizes the design flow of the Mutating Runtime Architecture. In Sect. 8.4 we detail the various design decisions and demonstrate the advantages of the proposed approach by hardening an AES block cipher. Finally, Sect. 8.5 concludes the chapter.

8.2 Mutating Runtime Architecture

The ongoing research in the area of computer-aided engineering (CAE) has meanwhile developed various techniques to map a formally denoted algorithm to an integrated circuit design. High-level synthesis is one of the core concepts in such a design flow. It consists of the following fundamental design activities in order to implement an algorithm in hardware: Allocation, Binding, and Scheduling.

First, the elementary operations needed to execute the algorithm are identified during Allocation. Each such operation is then mapped to one or two multiple types of suited hardware processing units. An explicit assignment of a specific task of the algorithm to a unit is addressed by Binding. Finally, Scheduling is required to order tasks in the time domain. Thus, by performing this activity the designer decides on the number of operations to be executed in parallel or on the sequence of data-independent operations. The resulting architectural properties of the circuit rely directly on the results of the above-mentioned basic activities of the CAE process, cf. [6]. Thereby, physical characteristics of the implemented algorithm may be affected significantly. Please note that design decisions in one of these activities in general do influence decisions in the other ones as well.

8.2.1 Design Properties

The concept of the Mutating Runtime Architecture (MRA) exploits the intrinsic influence of the mentioned design activities on the architecture in order to enable a flexible design space exploration. This exploration results in various feasible implementations of the same cryptographic algorithm, but with quite different physical properties. The challenge resulting from this approach is to identify the fundamental design properties which can be controlled individually by the corresponding design activities.

The outcome of Allocation specifies which *types of processing units* are required by the algorithm and thus denotes the first design property. Type of processing unit inherits its parameters from Allocation, but it is nearly independent from Binding and Scheduling. Thus, it is possible to replace an individual processing unit at runtime by means of a novel method named OnlineAllocation. The second property is the *degree of parallelism*, which inherits its design parameters from Binding. Independently from the type of the allocated processing unit, this degree can be varied based on the amount of available units. However, it can only be modified if the mutating method named DynamicBinding detailed in the sequel does not violate task dependencies. The last property is linked to Scheduling and addresses the *progression of tasks*, which are to be executed sequentially in a data-dependent order. The data flow is independent from the utilized processing units from a functional point of view. The related method FlexibleScheduling thus deals with task progression, whereas the overall task management shall be addressed by a middleware approach. Figure 8.1 visualizes the relationships of design activities, properties, and dedicated countermeasure construction methods.

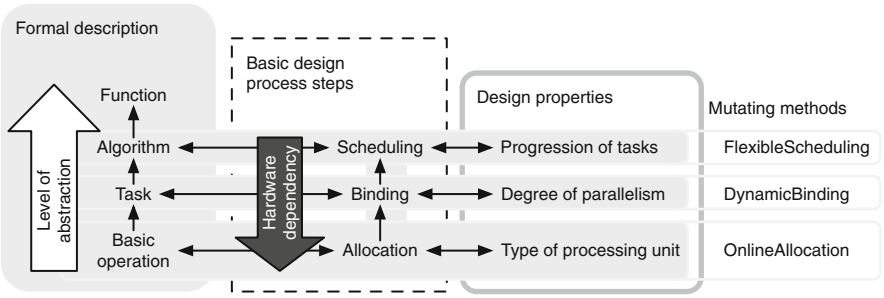


Fig. 8.1 Layer model for design activities, properties, and mutating methods

Hardware/Software co-design seems to be a feasible fundamental approach according to the discussion outlined above. Both task progression management and dynamic binding for the selected degree of parallelism are much easier to handle in software. Only the online allocation of different types of processing units cannot be assigned to the software partition due to their inherent hardware dependency thus resulting in a tailored HW/SW architecture on top of the envisaged FPGA implementation platform.

8.2.2 Online Allocation Method

The overall power consumption of CMOS circuits results to a large extent from the switching activity of the circuit. More specifically speaking, the various internal modules, interconnections, and configuration settings of LUTs of the FPGA platform are the origin of the characteristic controlled transient effects or glitches. Therefore, different implementations of the same basic operation within a cryptographic algorithm clearly influence the exploitable power consumption signature because of their also distinct switching activities. The statistical properties of the data-dependent power consumption in terms of mean and variance are thereby strongly affected by the characteristics of the glitches of the circuit. In case of a static design this property is relatively vulnerable to passive side-channel attacks, cf. [5, 11, 12, 21]. These attacks exploit the unique distribution of the power consumption signature. One possibility to compare the power consumption signature of different design variants featuring the same functionality is to produce a scatter plot like in Fig. 8.2.

In this figure we visualize the power consumption of four different designs of an SBox operation. Each of these circuits consists of one 8-bit register and one of the four different 8-bit AES SBox designs known as COMP, PPRM1, PPMR3, and TBL, respectively. On the x-axis the power consumption value of SBox design A at point in time t is displayed. The y-axis marks the related value of SBox design B. This example clearly indicates the difference in the exploitable power consumption

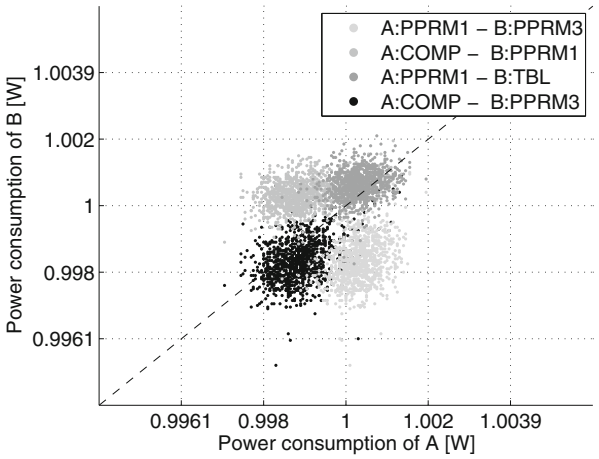


Fig. 8.2 Scatter plot of the power consumption of different SBox design variants

signature of design variants. Both variant-specific characteristics, i.e., the LUT configuration and the interconnect architecture of the LUT on the FPGA platform, cause different distributions of the power consumption, whereas the distribution strongly depends on the switching activity. This behavior may well be exploited as a countermeasure because processing units featuring identical functionality but different signatures help to minimize the exploitable power characteristics. However, one has to take into account that a low overall total correlation value between variants is a necessary, but not a sufficient condition to generate a complex distribution based on switching randomly between the individual distributions of different types of units. A more appropriate metric is the total correlation value of Hamming distance classes between different design variants of the same processing unit as proposed in [19].

The FPGA technology offers many techniques to take the concept of online allocation to practice. So, the exchange of processing unit types may be applied at various places. First, the randomization can be done within the switching network. Second, it may be applied on platform level by exploiting the partial reconfiguration features of an FPGA. These techniques differ in the required amounts of resource consumption and total power consumption as well as of design time due to the resulting device complexity and the available tool support. One of the first approaches to replace active units in order to manipulate the overall power consumption signature is reported in [1, 2]. There, Benini et al. proposed to utilize two different implementations of a processing unit and to randomly switch between them by means of a multiplexor. Compared to this work, these authors do not quantify the power characteristics in terms of statistical properties.

Switching Network A straight-forward approach to reshape the data path at runtime is to exploit a switching network based on multiplexors. The resulting advantage is the small amount of time needed to mutate the data path behavior in terms of its dynamic power consumption. The main drawbacks, however, result

from a considerable enlargement of the data path depth and from a decreased throughput. A substantial improvement in terms of resource consumption and throughput is presented in [10]. Here, instead of switching between two complete processing units to mutate the data path behavior, the randomization addresses directly the elementary components of the units. A lot of resources on the primitive layer of the FPGA platform can thus be shared and the unavoidable data path depth incrementation is under better control. To reach this goal, Madlener et al. in [10] propose a novel hardened multiplier for GF(2^n) aimed to public-key ECC applications denoted as "enhanced Multi-Segment-Karatsuba" (eMSK), which is an improvement of the Multi-Segment Karatsuba (MSK) multiplier introduced in [4]. The main advantage of the eMSK scheme is that it combines the efficiency of the recursive Karatsuba with the flexibility of the MSK scheme. Therefore, the order of the operations in the sequence strongly affects the power consumption due to a randomization in both time and amplitude domain resulting in an efficient countermeasure. Consequently, the varying order of the basic multiplication has a strong impact on the glitches in the combinatorial multiplier logic as well as on the bit flips in the accumulator. A similar countermeasure applied to the AES block cipher is proposed in [7], where the technique of composite-field-based SBoxes is exploited in order to switch between different variants. Such a concept was first investigated in [3]. Compared to many other masking schemes such as of, e.g., [17], the data path in [7] does not need to be doubled thus making this countermeasure suitable even for resource constrained designs.

Partial Reconfiguration Some RAM-based FPGA platforms such as Xilinx Virtex 5 support dynamic partial reconfiguration, a feature which may be exploited for the purpose of mutating data paths too. Instead of using more active reconfigurable resources for processing units operating in parallel, partial reconfiguration can be applied to save on resources. The data path depth increases not that much compared to a switching network and the resulting total power consumption is lower.

8.2.3 DynamicBinding Method

The property *degree of parallelism* influences the time-dependent amount of noise due to concurrently operating processing units. Therefore, the variance of the power consumption at a certain point in time is being manipulated. Variance manipulation strongly compromises the learning phase of a template attack, in which an attacker tries to establish a multivariable Gaussian-based model for the noise distribution. In addition, a dynamic activation of several processing units working in parallel also affects the noise characteristics of the measured power consumption. One efficient method to provide a dynamic binding of resources stems from the concept of virtualizing hardware components. Virtualization may be seen as an abstraction layer, which changes during runtime the link between a processing unit and an assigned task. Therefore, different tasks, which are bound to one or two multiple

Algorithm 8.3: RanParExec algorithm

Require: a given set of n processing units of m different types $PU_{set} = \{PU_1^{types}, PU_2^{types}, \ldots, PU_n^{types}\}$ with $PU_k^{types} = \{PU^1, PU^2, \ldots, PU^m\}$, $1 \leq k \leq n$ and two equal distributed random variables $R_1 = \{1, 2, \ldots, m\}$, $R_2 = \{1, 2, \ldots, n\}$

```
1: for 1≤ w ≤ u do
2:     select a realization of the random variable r_{1,w} ← R_1
3:     assign each op_{PU,w} to the processing unit type PU^{r_{1,w}}
4: end for
5: repeat
6:     repeat
7:         select a realization of the random variable r_{2,w} ← R_2
8:     until r_{(2,w)} ≤ number of currently in parallel executing op_{PU,w+}
9:     Execute the following k operations in parallel
10:    for 1 ≤ k ≤ r_{(2,w)} do
11:        execute op_{PU,k} with the assigned processing Unit PU
12:    end for
13:    w=w+r_{(2,w)}
14: until w > u
15: return
```

resources, can be shared or reordered. This kind of context switching on a processing unit may be handled transparently and does not need any modification of the scheduled task sequence within a cryptographic algorithm.

Random Concurrent Binding In this approach the number of units running in parallel varies as well as the binding of an operation to a specific unit. Therefore, several random combinations of unit types as provided by the method OnlineAllocation featuring different degrees of parallelism and bindings may thus be generated. Algorithm 8.3 named RanParExec illustrates this randomization technique denoted in pseudocode.

Virtualization Hardware virtualization is a method to abstract and decouple the executing hardware platform from the allocated algorithms. The virtualization technique presented in [20] exploits hardware components of the underlying FPGA platform and is taken as the basis for the approach of this work. A virtualized component may be shared by one or more tasks of a procedure or even of a complete algorithm. This component is addressed individually by the utilizing task set. But this task set allocates the hardware resource without any knowledge on whether it is being shared with a disjoint set of tasks.

Middleware Concept The application of virtualization between the executing processing units and the sequence of tasks based on the control and data flow can be established by an intermediate layer. It features almost the same functionalities as the so-called Middleware, a commonly used concept in the distributed computing domain. Thus, this concept may be viewed as an additional executive layer to

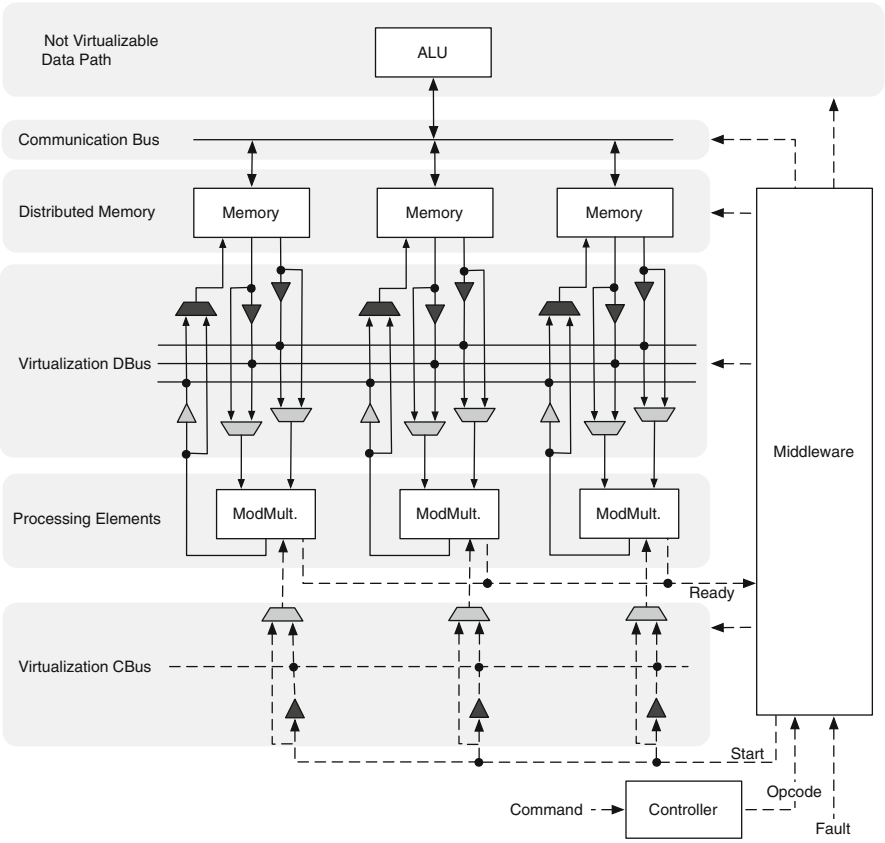
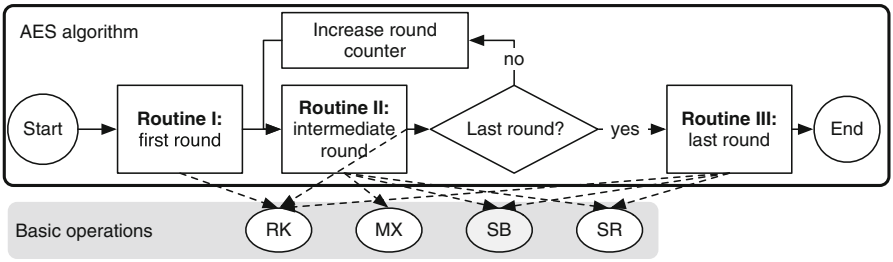


Fig. 8.3 Virtualized public-key ECC cipher architecture [20]

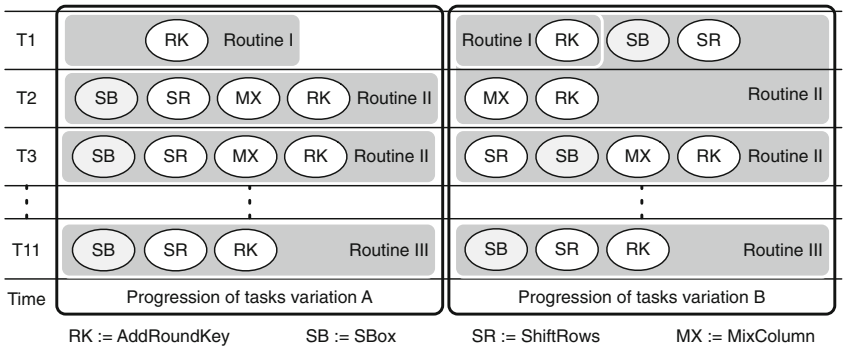
apply the method of virtualization for the outlined DynamicBinding method and has the administrative duty to organize the required links between tasks and executing resources. Figure 8.3 illustrates the outlined combined hardware virtualization and middleware approach applied to an ECC cipher architecture.

8.2.4 FlexibleScheduling Method

The last design property of Fig. 8.2 named *progression of tasks* is being addressed by FlexibleScheduling, which is the final method introduced to further enhance the MRA by additional countermeasures. FlexibleScheduling is thus intended to directly change the execution behavior of the tasks within the cryptographic algorithm during runtime, which affects considerably the physical behavior of the implementation. The impact on the power consumption signature due to progression



(a) Control flow of the AES algorithm



(b) Example of rearranged basic operations in a routine

Fig. 8.4 Rearrangement of operations by routine (re)shaping within AES. (**a**) Control flow of the AES algorithm. (**b**) Example of rearranged basic operations in a routine

of tasks is comparable to the commonly used method of shuffling independent operations in a cryptographic algorithm, if the granularity of processed operations per clock cycle does not change. Therefore, FlexibleScheduling decomposes a routine such that only the sequence of data-independent basic operations within the routine is being manipulated. The increased analysis effort for attacking a cryptographic module, which shuffles its data-independent operations, is well known [13]. Some interesting effects of FlexibleScheduling are illustrated in Fig. 8.4.

Most cryptographic algorithms are constructed either from basic operations or exploit other cryptographic functions, which are in turn composed of such operations. For instance, most block cipher algorithms such as AES or PRESENT are round oriented schemes. As illustrated in Fig. 8.4a, each different round function of AES can be implemented in its own routine, which utilizes a set of basic operations. Each of the three routines can be manipulated in terms of its physical behavior by means of different basic operations executed in one clock cycle without needing to change neither the number of bits processed in parallel nor the circuit-level implementation of these operations. In this example the basic operation of the SBox is handled by a mutable processing unit, which can be reallocated online while the device is active. In addition, the number of SBoxes operating in parallel

may be modified by a dynamic binding. As the rearrangement example in Fig. 8.4b demonstrates, the first round of AES, which is one of the two most common targets of a power analysis attack, can be merged with some basic operations of the second one. Therefore, the number of operations executed within one clock cycle is modified from the original scenario depicted on the left-hand side of Fig. 8.4b to the rearranged routine shown on the right-hand side. The power consumption signature is being disturbed by the sum of Gaussian distributions caused by operations being active during one clock cycle over the observed set of experiments featuring a varying operational activity. Thus, the resulting complex joint distribution acts as an additional countermeasure because it combines effects stemming both from manipulations by means of DynamicBinding and from the exchange of unit types resulting from OnlineAllocation.

8.3 Design Flow

The overall design flow for a Mutating Runtime Architecture results from the outlined design strategy and is illustrated in Fig. 8.5. Starting from a given cryptographic algorithm, a HW/SW partitioning needs to be done first such that procedures dominating the control flow of the algorithm are assigned to the software partition and the basic operations are grouped to form the hardware partition. Tasks composed mainly from basic operations should be mapped to software too in order to better utilize the hardware resources in the data path.

The information clustered in the software partition forms the basis for establishing various schedules by utilizing FlexibleScheduling. This method can be used to decompose routines or to rearrange procedures within a routine too. In contrast, OnlineAllocation and DynamicBinding address the hardware partition. First, the basic operation executed on a processing unit with the most exploitable power consumption op_{PU} needs to be identified. The units aimed to run the other basic operations can be implemented directly on the target FPGA as depicted in the lower part of Fig. 8.5. The generated schedules are then analyzed in order to identify the amount of op_{PU} usage. After this step, DynamicBinding produces the virtualization schemes. They are utilized in the sequel to construct the related virtualization module. The processing of the schemes for OnlineAllocation, FlexibleScheduling, and DynamicBinding can be conducted concurrently.

In the end, after these construction methods have jointly generated multiple countermeasures, all required components are assembled into the MRA featuring a virtualization module and various types of processing units. The randomization of the active processing units is handled directly by the middleware, i.e., the virtualization module, via a switching network or, if applicable, by exploiting a partial reconfiguration of the target FPGA platform.

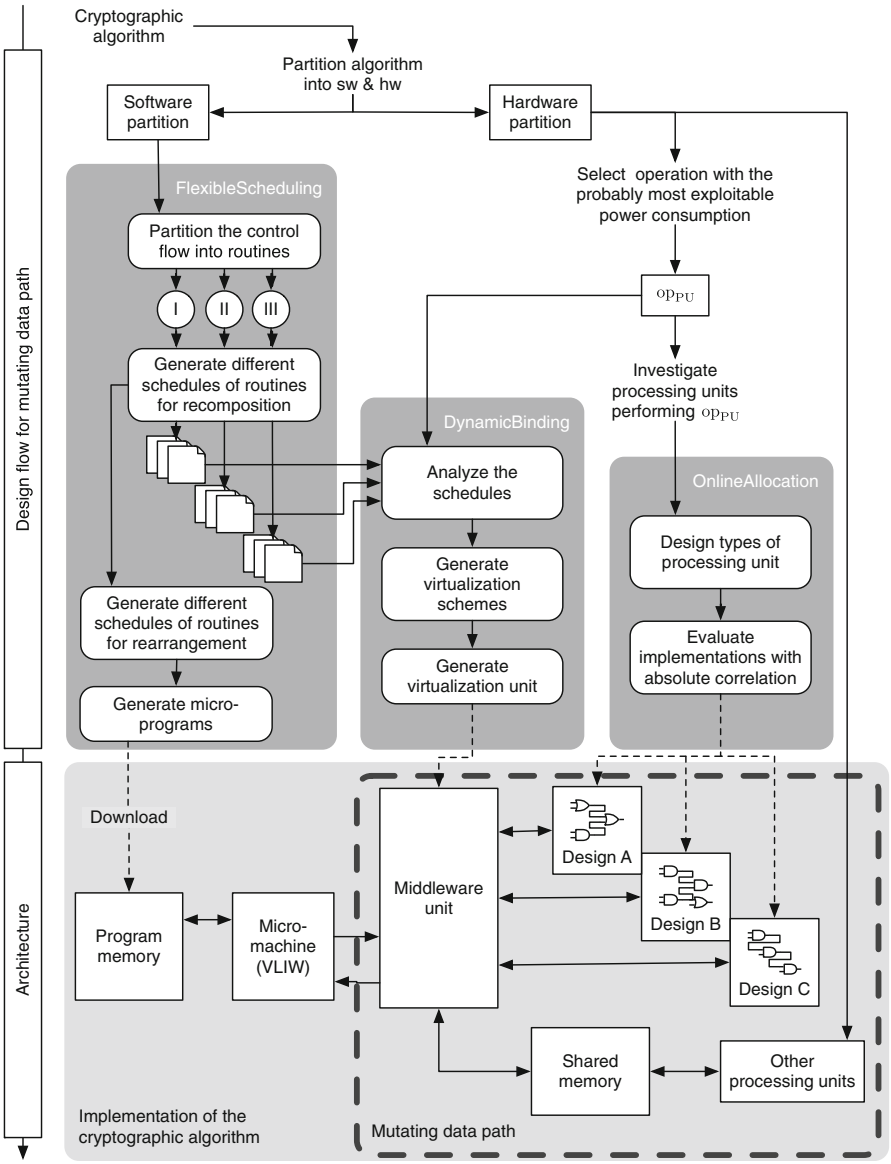


Fig. 8.5 Design flow for and outline of the Mutating Runtime Architecture

8.4 Case Study: Block Cipher AES 128-Bit

In this section we present the application of the proposed concept on the well-known symmetric-key AES algorithm. The hardened design will from now on be referred to as AES Mutate. One property of the advocated MRA is the manipulation

of in-parallel processed data, therefore the envisaged architecture of AES Mutate shall support a round-based as well as a non-round-based implementation style. The four basic operations of AES need to be modularized into efficient hardware components in order to support a 128-bit as well as some smaller word width processing. Therefore, the key scheduler has to be able to produce a complete new round key every clock cycle. An additional port is needed to attach a source of entropy in order to enable to mutate the block cipher non-deterministically.

8.4.1 Partitioning of the AES Modules

The generic MRA outlined in the lower part of Fig. 8.5 needs now to be adapted and refined according to the requirements of AES Mutate. An overview of the HW/SW partitioning of AES Mutate and the role of construction methods are illustrated in Fig. 8.6. As visible from this figure the software partition located on the left-hand side controls both the data flow within and the configuration of the cipher. Therefore, from a controlling point of view, the behavior in terms of reconfiguring the system and executing an encryption task is easy to adapt. The VLIW opcode for the configuration of the data path as well as the opcode to control the basic operations for an encryption are implemented in hardware and are stored in the program memory. A RAM module of the FPGA platform can be utilized for this purpose. Then, a tiny micro-machine addresses this memory in the scheduled order. The functionality of the middleware is separated into a software part, embedded into the micro-machine, and into a hardware part. The software part of the middleware manages the control flow and the execution order in case of changing the binding. The hardware part consists of the reconfigurable data path, whereas the basic

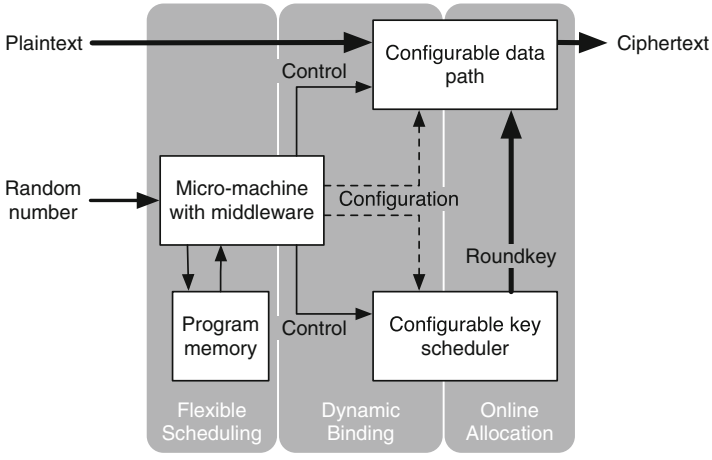


Fig. 8.6 Countermeasure construction methods and HW/SW partitions of AES Mutate

operations for executing a round operation are embodied as hardware components. The required operations for expanding the secret key according to the round key are grouped into the hardware partition too. The hardware partition of the key scheduler located on the lower right-hand side of Fig. 8.6 is reconfigurable too, so that the variation of the round execution in terms of processing intermediate values in parallel or of execution length, respectively, does not affect the computational correctness of the result.

The design-specific reasons for partitioning the functionalities of the mutating procedure in this way are directly visible from Fig. 8.6. The interface between the construction methods FlexibleScheduling and OnlineAllocation provides the DynamicBinding functionality. This functionality in turn is realized jointly by components of the micro-machine and by architecture entities within both the data path and the key scheduler. More specifically speaking, middleware components are used to put in place DynamicBinding as the central modification instance according to the design flow of Fig. 8.5.

8.4.2 Implementation

In the sequel the implementation procedure of the block cipher is being detailed. First, the attack threat on AES is discussed in order to identify, which basic operation is most vulnerable due to its power consumption signature and thus has to be secured. Based on this discussion the different properties of the mutating architecture are reasoned in order to identify the parts of the data path, which have to be reconfigurable to increase the effort of a power analysis attack. The development of the cipher follows the generic design flow and exercises all proposed construction methods in order to harden the design by multiple countermeasures.

8.4.2.1 Design Requirements

The mutating architecture of the block cipher shall fulfill the following requirements and features:

- Executing of the SBox operation on quite different implementations
- Dynamically changing the degree of in-parallel processing SBox units
- Merging round operations into one clock cycle
- Manipulating the word width being processed during a clock cycle
- Randomizing the state representation in the shared memory.

The SBox operation is selected as the target of OnlineAllocation as well as for DynamicBinding, because it is the most vulnerable operation in two of three round operations. In the studied attack scenarios in [19], the SBox operation was the central element to determine the estimated exploitable power consumption of the whole block cipher. It is a bijective operation with a high amount of diffusion and,

as reported in [15], it has the highest power consumption of all the basic operations of the AES. There is a wealth of various kinds of implementations of SBox units reported in literature featuring different design-specific characteristics, cf. [15, 18], which may be exploited to manipulate the power consumption of the operation.

Due to the higher power consumption of in-parallel processing SBox variants the variance of the power signature is enlarged. Thus, the SNR of extractable information within the captured power traces will decrease and thereby increase the attacking effort. Varying the number of operations per clock cycle and merging round operations (routines of the AES, cf. Fig. 8.4) are means to hide the amount of concurrently active processing SBoxes. In this context FlexibleScheduling may be seen as a combination of Shuffling [13] and Temporal Jitter countermeasures [14]. The last point in the list above is intended to prevent an exploitation of the XOR operation by kind of a randomized Register Pre-charging and Shuffling.

8.4.2.2 Data Path Architecture

Figure 8.7 provides an overview on the structure of the data path. As mentioned before the data path for the transformation of the plaintext to the ciphertext (upper circuit part) and the data path for the key expansion (lower circuit part) have to be mutable. The required degree of in-parallel data processed per clock cycle is scalable in byte-wise steps from 8 bit up to 128 bit. The key scheduler circuit is extended by a multiplexor in order to switch between either the updated bytes of the round key value or the previous bytes of the round key. Thus, the key scheduler can process in-parallel the next round key on 128 bit at once or in 96, 64, or 32 bit wide chunks.

We selected a set of AES SBox implementations with different physical behavior and power signatures for usage in the P-units of Fig. 8.7. For the choice of SBox types we followed the recommendation in [19] due to the outlined evaluation and a mature selection process. In order to save space on the FPGA platform, a concurrent SBox management is in place to exchange the SBox type of each byte section of the AES: The virtualization method discussed above is used to change the binding between the state register and the SBox implementations in addition to partial reconfiguration. So, not every SBox in the data path needs to be partially reconfigurable as depicted in Fig. 8.5. This design decision helps to save resources and execution time.

The challenge of changing the degree of in-parallel processed data through the SBox unit stems from the entangled data dependency between columns and rows of the AES round state matrix. The SBox is embedded in the so-called P-unit, which also offers the functionality of the AES specific basic operations MixColumn and AddRoundkey as well as a bypass operation. The P-units work on the columns of the matrix and they are interconnected by a reconfigurable ShiftRows (RSR) function in order to process the complete state matrix. The RSR can shift each byte of a row from one byte to the right up to two bytes on the left as well as bypass the shift operation in order to maintain the word position of the four bytes in the rows. This unit can also be used to either shuffle the state representation on-the-fly

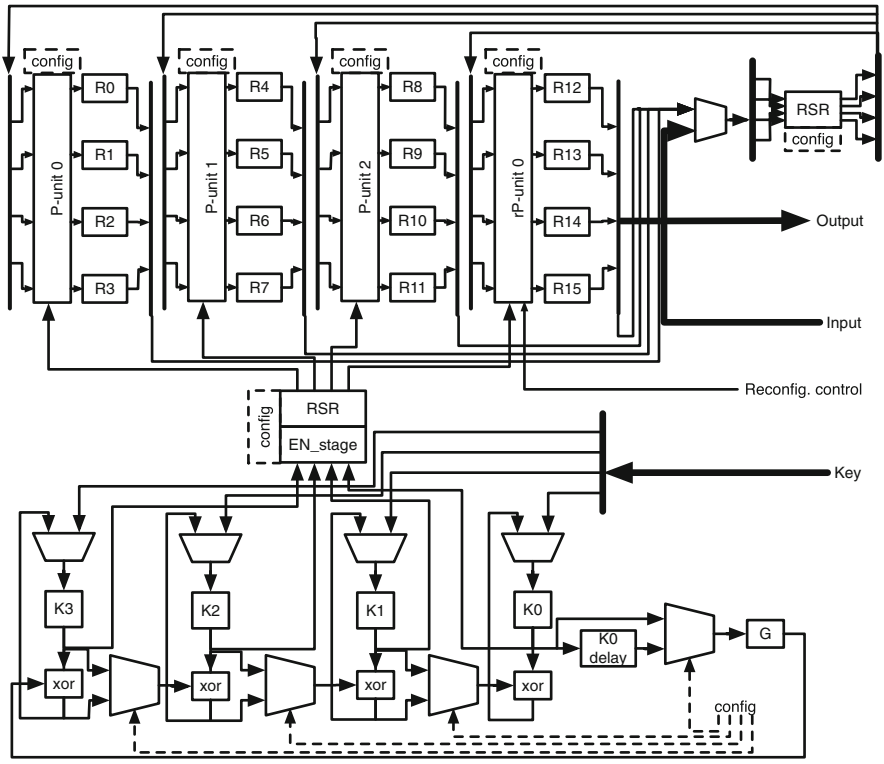


Fig. 8.7 Schematic of the AES Mutate data path

or to change the binding to a specific SBox instance. In case of shuffling the state, the subkeys have also to be shuffled in the same order. Thus, an additional RSR unit is located between the P-units and the key scheduler. Therefore, the state randomization can be inserted within a round operation without needing any additional clock cycles for the reconstruction of the original value of the state register position.

The AddRoundkey function is merged with the MixColumn function to the so-called *MXOR*, which is configurable so as to perform the MixColumn operation followed by an XOR operation for the AddRoundkey functionality or just to execute an XOR. Thus, the P-unit provides three functionalities, which can either be performed in a combination at any level or may be executed individually,[1] and a bypass functionality. In order to support the partial reconfiguration on the Virtex 5 platform at moderate costs in terms of resources and execution time, the so-called reconfigurable P-unit (*rP-unit*) is added to this data path instead of a forth P-unit,

[1] SBox, MixColumn, AddRoundkey.

cf. Fig. 8.7. The rP-unit features an additional layer of multiplexors aimed to route data between the partial reconfigurable areas of the SBox. While one reconfigurable area is being updated with a new configuration, the other one continues to execute the SBox operation.

8.4.2.3 Virtualization Scheme

As mentioned in Sect. 8.4.1, the data flow routing functionality of the middleware is embedded in the data path. A virtualization of this design can be conducted by utilizing the RSR units to reroute or to shuffle the content of the register states. Thereby, different SBox implementations may be utilized for each byte-width entry of the AES state matrix in case each P-unit/rP-unit instance features SBox implementation variants. The configuration settings of the P-unit/rP-unit are the second control element supporting the virtualization in order to realize the method DynamicBinding. Due to these settings the amount of in-parallel working SBoxes per clock cycle can be controlled.

As visible from Fig. 8.6 a micro-machine is part of AES Mutate, which executes a tailored VLIW opcode command set for controlling the configuration settings of the P-units/rP-unit as well as of the RSR units. Due to a flexible VLIW programming environment and the associated hardware architecture of the middleware this functionality is embedded into the data path. The outlined virtualization provides a suitable interface between the DynamicBinding and FlexibleScheduling methods. The configuration settings for a P-unit/rP-unit can be used in addition to modify the amount of in-parallel working hardware components. For instance, the execution of multiple XOR operations can be manipulated in terms of the degree of in-parallel working XORs as well as the amount of basic AES operations executed within one clock cycle.

8.4.2.4 Merging Round and Routines

The modularized, flexible, and reconfigurable HW/SW architecture of AES Mutate is able to randomize the execution of every basic AES operation, because an application of FlexibleScheduling offers the ability to modify the degree of in-parallel working processing units of all basic AES operations. The width of the data path is adaptable for all operations resulting in 32, 64, 96, and 128 bit, respectively, processed within one clock cycle. The layout of the micro-machine in combination with the embedded middleware infrastructure present in the data path provides a flexible rescheduling feature via VLIW operations, which are independent of the current data path processing width setting. Therefore, the number of data-dependent operations processed within one clock cycle is randomizable and, thus, basic operations of different AES round operations can be merged.

The rescheduling of the AES encryption results from utilizing small micro-programs, which contain just one or few 36 bit wide VLIW opcode commands. Each round or merged round operation is represented by such a micro-program.

Within one clock cycle the opcode controls the processing units of the basic AES operations and sets the configuration of the data path. Thus, the variable word width of the data path is defined by just an opcode command.

The MSB bit of the opcode is used as a delimiter in order to identify the end of a round operation. Therefore, a complete AES encryption can be composed from randomly selecting different micro-programs for each round operation of the AES algorithm. The lower 32 bits of the opcode are used to reconfigure the data path during runtime in order to randomize the overall power consumption. These micro-programs are therefore more or less reconfiguration settings aimed to change dynamically the architecture of the data path. This context-based reconfiguration approach was chosen because the partial reconfiguration scheme offered by the Xilinx, Inc. design flow takes from our perspective too much time to complete. Therefore, we decided to aim for a reasonable tradeoff by using only one rP-unit next to three P-units and a modular micro-programming approach instead.

8.4.3 Side-Channel Analysis Results

We compare the results gained from the hardened cipher to those of an unprotected implementation in order to emphasize the additional effort of attacking the secured design by means of a passive power analysis. The unprotected implementation is a round-based operating AES scheme implementation applying the COMP SBox variant and is denoted as AES COMP. The resulting block cipher processes 128 bits per clock cycle. The evaluation was conducted with 500,000 power traces in total, which were captured from the SASEBO-GII board. In case of the analysis of the AES Mutate design an additional trivium stream cipher [16] is introduced aimed to produce the random input data.

For a comparison of the side-channel resistance properties of AES Mutate and AES COMP the leakage of the SBox computation of the first round was analyzed. As analysis method we selected the Stochastic Approach [8], because it is a powerful profiling method that exploits the leakage at best by means of a self-learning linear regression model. We analyzed 450,000 power traces with random plaintext in the profiling phase to build the linear model. Then we captured 50,000 additional traces from the same device for the attack phase. Using the same device for both the profiling and the attack phase of a template-based analysis represents the optimal case for an attacker and, hence, should yield as the evaluation result the lower bound of security in terms of side-channel resistance. As distinguisher and evaluation metric for detecting exploitable leakage we applied the SNR-metric introduced in [19].

Figure 8.8 depicts the results of the side-channel analysis of these implementation variants on top of the Xilinx Virtex 5 platform. The figure visualizes the amount of correctly revealed secret key bytes over the number of required traces during the attack phase. One can easily derive from this figure that the first correct subkeys extracted from the unprotected AES COMP design were unveiled after just a few hundred traces and all secret subkeys were recovered after using 17,780 traces in total. The attack result of the AES Mutate design hardened by multiple

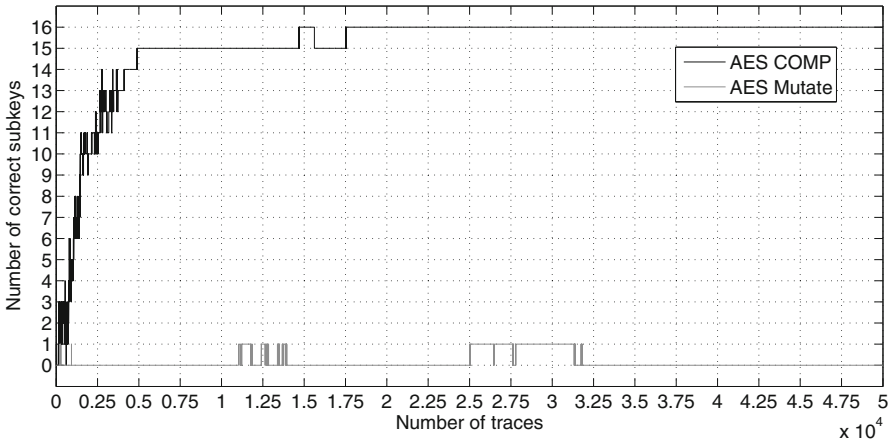


Fig. 8.8 Comparison of the analysis results of AES COMP and AES Mutate ciphers

countermeasures shows in contrast an unstable extraction of just one out of 16 subkeys. Hence, the secret key keeps its strength during the attack phase for the complete set of applied traces. The unstable analysis result nicely indicates that the advocated concept of mutating cipher architectures works very well in order to generate a wide noise distribution intended to protect the real key-characteristic distribution.

8.5 Summary

We presented in this chapter a novel technique to secure cryptographic modules of embedded systems against power analysis attacks, which form the most popular class of passive, non-invasive SCA strategies. Taking the inherent reconfiguration properties of FPGAs and the generic design methodology of computing systems as foundations, we first introduced construction methods for multiple countermeasures and presented subsequently a Mutating Runtime Architecture, which is well-suited to harden quite different cipher types. We then detailed as proof of concept a case study aimed to secure an FPGA implementation of the AES algorithm and highlighted the advantage of an automatic generation of multiple countermeasures.

Acknowledgements This work was funded in part by DFG, the German science foundation, under grant no. HU620/12 within the DFG Priority Programme 1148 “Reconfigurable Computing Systems” in cooperation with CASED.

References

1. L. Benini, A. Macii, E. Macii, E. Omerbegovic, M. Poncino, F. Pro, A novel architecture for power maskable arithmetic units, in *GLSVLSI* (ACM, New York, 2003), pp. 136–140
2. L. Benini, A. Macii, E. Macii, E. Omerbegovic, F. Pro, M. Poncino, Energy-aware design techniques for differential power analysis protection, in *DAC* (ACM, New York, 2003), pp. 36–41
3. D. Canright, A very compact Rijndael S-Box. Technical Report, Naval Postgraduate School (2005)
4. M. Ernst, M. Jung, F. Madlener, S.A. Huss, R. Blümel, A reconfigurable system on chip implementation for elliptic curve cryptography over GF(2^n), in *CHES*. Lecture Notes in Computer Science, vol. 2523 (Springer, Berlin, 2002), pp. 381–399
5. W. Fischer, B.M. Gammel, Masking at gate level in the presence of glitches. in *CHES*, ed. by J.R. Rao, B. Sunar. Lecture Notes in Computer Science, vol. 3659 (Springer, Berlin, 2005), pp. 187–200
6. D.D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification*, 1st edn. (Springer, Berlin, 2009)
7. B. Jungk, M. Stöttinger, J. Gampe, S. Reith, S.A. Huss, Side-channel resistant AES architecture utilizing randomized composite-field representations, in *FPT* (IEEE, New York, 2012), pp. 125–128
8. M. Kasper, W. Schindler, M. Stöttinger, A stochastic method for security evaluation of cryptographic FPGA implementations, in *FPT* ed. by J. Bian, Q. Zhou, P. Athanas, Y. Ha, K. Zhao (IEEE, New York, 2010), pp. 146–153
9. P.C. Kocher, J. Jaffe, B. Jun, Differential power analysis, in *CRYPTO 99*, ed. by M.J. Wiener. Lecture Notes in Computer Science, vol. 1666 (Springer, Berlin, 1999), pp. 388–397
10. F. Madlener, M. Stöttinger, S.A. Huss, Novel hardening techniques against differential power analysis for multiplication in GF(2^n), in *FPT* (IEEE, New York, 2009)
11. S. Mangard, T. Popp, B.M. Gammel, Side-channel leakage of masked CMOS gates, in *CT-RSA*, ed. by A. Menezes. Lecture Notes in Computer Science, vol. 3376 (Springer, Berlin, 2005), pp. 351–365
12. S. Mangard, N. Pramstaller, E. Oswald, Successfully attacking masked AES hardware implementations, in *CHES*, ed. by J.R. Rao, B. Sunar. Lecture Notes in Computer Science, vol. 3659 (Springer, Berlin, 2005), pp. 157–171
13. S. Mangard, T. Popp, M.E. Oswald, *Power Analysis Attacks - Revealing the Secrets of Smart Cards* (Springer, Berlin, 2007)
14. N. Mentens, B. Gierlichs, I. Verbauwhede, Power and fault analysis resistance in hardware through dynamic reconfiguration, in *CHES*, ed. by E. Oswald, P. Rohatgi. Lecture Notes in Computer Science, vol. 5154 (Springer, Berlin, 2008), pp. 346–362
15. S. Morioka, A. Satoh, An optimized S-box circuit architecture for low power AES design, in *CHES*, ed. by B.S.K. Çetin Kaya Koç Jr., C. Paar. Lecture Notes in Computer Science, vol. 2523 (2002), pp. 172–186
16. C. Paar, J. Pelzl, *Understanding Cryptography - A Textbook for Students and Practitioners* (Springer, Berlin, 2010)
17. F. Regazzoni, W. Yi, F.X. Standaert, FPGA implementations of the AES masked against power analysis attacks, in *COSADE* (2011), pp. 55–66
18. A. Satoh, S. Morioka, K. Takano, S. Munetoh, A compact Rijndael hardware architecture with S-Box optimization, in *ASIACRYPT*, ed. by C. Boyd. Lecture Notes in Computer Science, vol. 2248 (Springer, Berlin, 2001), pp. 239–254
19. M. Stöttinger, Mutating runtime architectures as a countermeasure against power analysis attacks. PhD thesis, Technische Universität Darmstadt (2012)

20. M. Stöttinger, A. Biedermann, S.A. Huss, Virtualization within a parallel array of homogeneous processing units, in *ARC*, ed. by P. Sirisuk, F. Morgan, T.A. El-Ghazawi, H. Amano. Lecture Notes in Computer Science, vol. 5992 (Springer, Berlin, 2010), pp. 17–28
21. T. Sugawara, N. Homma, T. Aoki, A. Satoh, Differential power analysis of AES ASIC implementations with various S-box circuits, in *ECCTD* (IEEE, New York, 2009), pp. 395–398

Part IV
Security and Trust Validation

Chapter 9
Validation of IP Security and Trust

Farimah Farahmandi and Prabhat Mishra

9.1 Introduction

Hardware Trojans are small malicious components added to the circuit. They are designed in a way that they are inactive most of the run-time, however, they change the functionality of the design or defeat the trustworthiness of it by leaking valuable information when they are triggered. Hardware Trojans can be categorized into two architectural types: combinational and sequential. A combinational Trojan is activated based on specific conditions and assignments on a portion of internal signals or flip-flops. On the other hand, sequential Trojans are small finite state machines that are activated based on a specific sequence. Each hardware Trojan consists of trigger and payload parts. Trigger is responsible for producing (activating) a set of specific conditions to activate the undesired functionality. Payload part is responsible for propagating the effect of the malicious activity to the observable outputs.

There are a wide variety of Trojans and they may be inserted during different phases such as specification time by changing timing characteristics or in design time by reusing untrusted available IPs offered by third parties or in fabrication time by altering the mask sets. However, it is more likely that Trojans are inserted during design time than manufacturing time as the attacker has better understanding of the design and can design trigger conditions as extremely rare events.

It is difficult to construct a fault model to characterize Trojan's behavior. Moreover, Trojans are designed in a way that they can be activated under very rare conditions and they are hard to detect. As a result, existing testing methods are impractical to detect hardware Trojans. There are several run-time remedies

F. Farahmandi (✉) • P. Mishra
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA
e-mail: ffarahmandi@ufl.edu; prabhat@ufl.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_9

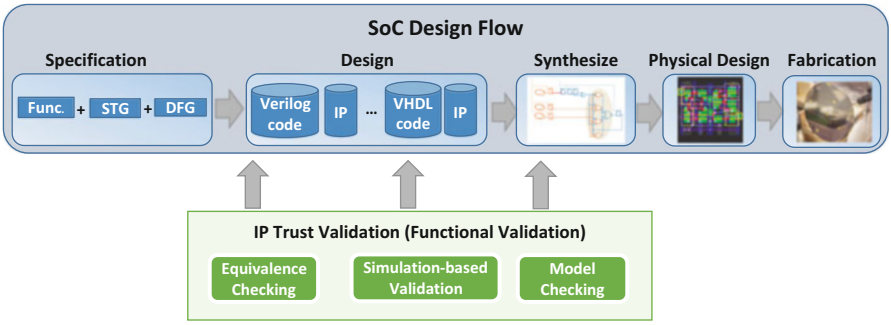


Fig. 9.1 Security validation flow. IPs used in SoC design should be validated against functional metrics using different methods such as equivalence checking, simulation-based validation, and model checking to insure security properties (nothing more, nothing less)

including disable backdoor triggers [27, 29], unused component identification [28], and run-time monitor of some design properties violence.

A major concern with the hardware IPs acquired from external sources (third party IP) is that they may come with deliberate malicious implants to incorporate undesired functionality, undocumented test/debug interface working as hidden backdoor, or other integrity issues. Therefore, security validation to identify untrusted IPs is critical part of the design process of digital circuits. Figure 9.1 shows that security validation should be done in different phases of SoC design to identify malicious functionality which may come from untrusted third party vendors.

There are several approaches focused on logic testing and pre-silicon validation efforts in order to detect and activate the potential hardware Trojans. These methods are described in Sect. 9.2. We review trust validation techniques based on equivalence checking in Sect. 9.3. Finally, model checking-based trust validation is outlined in Sect. 9.4.

9.2 Logic Testing for Trojan Detection

Several approaches are focused on generation of guided test vectors and compare the produced primary outputs with golden/expected outputs to detect and activate hardware Trojans. Traditional test generation techniques may not be beneficial as Trojans are designed in a way that they will be activated under very rare sequences of the inputs. In this section, we review simulation-based validation approaches including rare-node activation, redundant circuit detection, N-detect ATPG, and code coverage techniques as it is shown in Fig. 9.2.

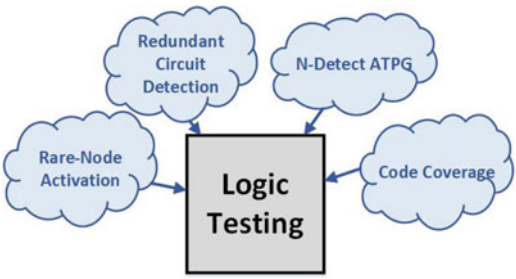


Fig. 9.2 Simulation-based validation for Trojan detection

9.2.1 Utilization Rarely Used Components for Trojan Detection

Unused Circuit Identification (UCI) algorithm is proposed by Hicks et al. [15] to report redundant components in pre-silicon verification time based on data flow dependencies of the design. The authors believed that these components can be used by an attacker to insert malicious functionality as they want to make sure that the inserted Trojan cannot be revealed during pre-silicon verification. To detect the hardware Trojans, the UCI algorithm identifies the signal pairs that they have the same source and sink first. Then, it simulates the HDL netlist with verification tests to find unused circuits that they do not affect on primary outputs and the UCI algorithm deactivates those unused pairs. However, the authors showed in their later work that there are many other types of malicious circuits that do not exhibit the hidden behavior and UCI algorithm cannot detect them [25].

Run-time solutions increase the circuit complexities and designers try to come up with efficient off-chip validation approaches to measure the trustworthiness of designs. Waksman et al. [28] proposed a pre-fabrication method to flag almost unused logic components based on their Boolean functions and their control values (FANCI). The authors believe that the almost-unused logic (rare nodes) will be the targets of attackers to insert hard-to-detect Trojans as the rare nodes have rarely impact on the functionality of the design outputs. They construct an approximate truth table (can also be mapped to vectors) for each internal signal to determine its effect on primary outputs. They use different heuristics to identify backdoors from control value vectors. However, this method reports 1–8 % of the total nodes of a design as potential malicious logic and it may incorrectly flag many as safe gates (false positives). Moreover, if the design is completely trustworthy, this method still reports many nodes as suspicious logic.

FANCI sets a probability threshold for trigger conditions and marks every signal that has lower activation probability than the pre-defined threshold as a suspicious signal. For example, suppose that the threshold is defined as 2^{-32}. If a Trojan trigger is a 32-bit specific value at a specific clock cycle, the probability of trigger activation is 2^{-32} and it is marked as malicious signal. DeTrust [32] has introduced a type of Trojans whose trigger happens across multiple clock cycles so the probability of their activation is computed higher and FANCI will report them as safe gates by mistake. For example, if the 32-bit trigger arrives in 8-bit chunks through 4

consecutive clock cycles, FANCI computes the trigger activation probability as 2^{-8} and because it is higher than the threshold (2^{-32}), it marks them as safe gates.

VeriTrust which is proposed by Zhang et al. [33] tires to find unused or nearly unused circuits which are not active during verification tests. The method classifies trigger conditions in two categories: bug-based and parasite-based hardware Trojans. Bug-based Trojan deviates the functionality of the circuit. On the other hand, parasite-based Trojans add extra functionality to the normal functionality of the design by introducing new inputs. In order to find the parasite-based trigger inputs, it first verifies the circuit by the existing verification tests. In the next step, it finds the entries of design functionality (like entries of Karnaugh-map) that are not covered during the verification. Then, it sets uncovered entries as don't cares to identify redundant inputs. DeTrust [32] introduced a type of Trojans where each of its gates is driven by a subset of primary inputs. Therefore, VeriTrust cannot detect this type of Trojan.

9.2.2 ATPG-Based Test Generation for Trojan Detection

In a recent case study [31], code coverage analysis and Automatic Test Pattern Generation (ATPG) are employed to identify Trojan-inserted circuits from Trojan-free circuits. The presented method utilizes test vectors to perform formal verification and code coverage analysis in the first step. If this step cannot detect existence of the hardware Trojan, some rules are checked to find unused and redundant circuits. In the next step, the ATPG tool is used to find some patterns to activate the redundant/dormant Trojans. Code coverage analysis is done over RTL (HDL) third party IPs to make sure that there are no hard-to-activate events or corner-case scenarios in the design which may serve as a backdoor of the design and leak the secret information [1, 31]. However, Trojans may exist in design that have 100 % code coverage.

Logic testing would be beneficial when it uses efficient test vectors that can satisfy the Trojan triggering conditions as well as propagate the activation effect to the observable points such as primary outputs. Therefore, the test can reveal the existence of the malicious functionality. These kinds of tests are hard to create since trigger conditions are satisfied after long hours of operation and they are usually designed with low probability. As a result, traditional use of existing test generation tools like ATPGs is impractical to produce patterns to activate trigger conditions. Chakraborty et al. proposed a technique (which is called MERO) to generate tests to activate rare nodes multiple times [6]. Their goal is to increase the probability of satisfying trigger conditions to activate Trojans. However, the effect of activation of trigger conditions caused by MERO tests can be masked as it does not consider payload when it generates the test. Saha et al. [23] proposed a technique based on genetic algorithm to guide ATPG tool in order to generate tests that not only activate Trojan triggers but also propagate the effect using payload nodes. Their goal is to observe the effect of the Trojan circuit from primary outputs.

A Trojan localization method was presented in [1]. The proposed method consists of four steps. In the first step, the design is simulated using random test vectors or tests generated by sequential ATPG tool to identify easy-to-detect signals. In the second step, full scan N-detect ATPG is employed to detect rare nodes. In the next step, an SAT solver is utilized to perform equivalence checking of suspicious netlist against the golden netlist to prune the search space of suspicious signals. Finally, uncovered gates are clustered using region isolation method to find the gates responsible for malicious functionality. However, as this approach uses SAT solvers, it fails for large and complex circuits. The method also requires a golden netlist which is usually hard to get.

Pre-defined templates are used to detect Trust-HUB benchmarks (https://www.trust-hub.org/) by Oya et al. [20]. First, a structural template library is constructed based on existing types of Trojans and their features. In the next step, the templates are scored based on their structures and whether these structures are observed only in Trojan circuits. Then a score threshold is defined to classify Trojan-inserted circuits from Trojan-free circuits. However, this approach may fail when an adversary inserts new Trojans.

9.3 Trojan Detection Using Equivalence Checking

Equivalence checking is used to formally prove two representations of a circuit design exhibit exactly the same behavior. From security point of view, verification of the correct functionality is not enough. The verification engineers have to make sure that no additional activity exists besides the normal behavior of the circuit. In other words, it is necessary to make sure that the IP is performing the expected functionality - nothing more and nothing less [13]. Equivalence checking is a promising approach to measure the level of trust for a design.

Figure 9.3 shows a traditional approach for performing equivalence checking using SAT solvers. The primary outputs of specification and implementation are fed to "xor" gates and the whole design (specification, implementation, and extra xor gate) are modeled as Boolean formulas (CNF clauses). If the specification and implementation are equivalent, the output of the xor gate should be always zero (false). If the output becomes true for any input sequence, it implies that the

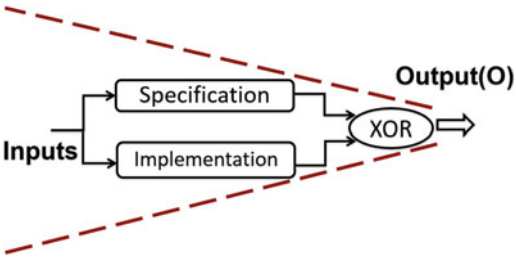


Fig. 9.3 Equivalence checking using SAT solvers

specification and the implementation are producing different outputs for the same input sequence. Therefore, the constructed CNF clauses of the input cone of "O" are used as an input of an SAT solver to perform equivalence checking. If the SAT solver finds a satisfiable assignment, the specification and implementation are not equivalent. SAT solver-based equivalence checking techniques can lead to state-space explosion when large IP blocks are involved with significantly different specification and implementation. Similarly, SAT solver-based equivalence checking approaches fail for complex arithmetic circuits with larger bit-widths because of bit blasting of arithmetic circuits.

9.3.1 Gröbner Basis Theory for Equivalence Checking of Arithmetic Circuits

A promising direction to address the state-space explosion problem in equivalence checking of hardware design is to use symbolic computer algebra. Symbolic algebraic computation refers to application of mathematical expressions and algorithmic manipulations methods to solve different problems. Symbolic algebra especially *Gröbner basis* theory can be used for equivalence checking and hardware Trojan identification as it formally checks two levels of a design and search for components that cause mismatch or change the functionality (hardware Trojans).

Computer symbolic algebra is employed for equivalence checking of arithmetic circuits. The primary goal is to check equivalence between the specification polynomial f_{spec} and gate-level implementation C to find potential malicious functionality. The specification of arithmetic circuit and implementation are formulated as polynomials. Arithmetic circuits constitute a significant portion of datapath in signal processing, cryptography, multimedia applications, error root causing codes, etc. In most of them, arithmetic circuits have a custom structure and can be very large so the chances of potential malfunction are high. These bugs may cause unwanted operations as well as security problems like leakage of secret key [3]. Thus, verification of arithmetic circuits is very important.

Using symbolic algebra, the verification problem is mapped as an ideal membership testing [10, 12, 24]. These methods can be applied on combinational [18] and sequential [26] Galois Filed $\mathbb{F}_{2^k}$ arithmetic circuits using Gröbner Basis theory [8] as well as signed/unsigned integer $\mathbb{Z}_{2^n}$ arithmetic circuits [10, 13, 30]. Another class of techniques are based on functional rewriting [7]. The specification of arithmetic circuit and implementation are converted to polynomials which constructs a multivariate ring with coefficients from $\mathbb{F}_{2^k}$. These methods use *Gröbner basis* and *Strong Nullstellent* over Galois field to formulate the verification problem as an ideal membership testing of specification polynomial f_{spec} in the ideal constructed by circuit polynomials (ideal I). Ideal I can have several generators, one of these generators is called Gröbner basis. First, Gröbner basis theory [8] is briefly described. Next, the application of Gröbner basis theory for security verification of integer arithmetic circuits is presented.

Let $M = x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$ be a monomial and $f = C_1M_1 + C_2M_2 + \cdots + C_tM_t$ be a polynomial with $\{c_1, c_2, \ldots, c_t\}$ as coefficients and $M_1 > M_2 > \cdots > M_t$. Monomial $\text{lm}(f) = M_1$ is called leading monomial and $\text{lt}(f) = C_1M_1$ is called leading term of polynomial f. Let $\mathbb{K}$ be a computable field and $\mathbb{K}[x_1, x_2, \ldots, x_n]$ be a polynomial ring in n variables. Then $< f_1, f_2, \ldots, f_s >= \{\sum_{i=1}^{n} h_i f_i : h_1, h_2, \ldots, h_s \in \mathbb{K}[x_1, x_2, \ldots, x_n]\}$ is an ideal I. The set $\{f_1, f_2, \ldots, f_s\}$ is called generator or basis of ideal I. If $V(I)$ shows the affine variety (set of all solution of $f_1 = F_2 = \cdots = f_s = 0$) of ideal I, $I(V) = \{f_i \in \mathbb{K}[x_1, x_2, \ldots, x_n] : \forall v \in V(I), f_i(v) = 0\}$. Polynomial f_i is a member of $I(V)$ if it vanishes on $V(I)$. Gröbner basis is one of the generators of every ideal I (when I is other than zero) that has a specific characteristic to answer membership problem of an arbitrary polynomial f in ideal I. The set $G = \{g_1, g_2, \ldots, g_t\}$ is called Gröbner basis of ideal I, if $\forall f_i \in I, \exists g_j \in G : \text{lm}(g_j) | \text{lm}(f_i)$.

The Gröbner basis solves the membership testing problem of an ideal using sequential divisions or reduction. The reduction operation can be formulated as follows. Polynomial f_i can be reducible by polynomial g_j if $\text{lt}(f_i) = C_1M_1$ (which is non-zero) is divisible by $\text{lt}(g_i)$ and r is the remainder ($r = f_i - \frac{\text{lt}(f_i)}{\text{lt}(g_j)}.g_j$). It can be denoted by $f_i \xrightarrow{g_j} r$. Similarly, f_i can be reducible with respect to set G and it can be represented by $f_i \xrightarrow{G}_+ r$.

The set G is Gröbner basis ideal I, if $\forall f \in I, f_i \xrightarrow{G}_+ 0$. Gröbner basis can be computed using Buchberger's algorithm [4]. Buchberger's algorithm is shown in Algorithm 9.1. It makes use of a polynomial reduction technique named S-polynomial as defined below.

Definition 1 (S-polynomial). Assume $f, g \in \mathbb{K}1, x_2, \ldots, x_n]$ are non-zero polynomials. The S-polynomial of f and g (a linear manipulation of f and g) is defined as: $\text{Spoly}(f, g) = \frac{\text{LCM}(\text{LM}(f), \text{LM}(g))}{\text{LT}(f) * f} - \frac{\text{LCM}(\text{LM}(f), \text{LM}(g))}{\text{LT}(g) * g}$, where $\text{LCM}(a, b)$ is a notation for the least common multiple of a and b.

Algorithm 9.1: Buchberger's algorithm [4]

```
1:  procedure GRÖBNER BASIS FINDER
2:      Input: ideal I =< f1, f2, ..., fs > ≠ {0}, initial basis F = {f1, f2, ..., fs}
3:      Output: Gröbner Basis G = {g1, g2, ..., gt} for ideal I
4:      G = F
5:      V = G × G
6:      while V ≠ 0 do
7:          for each pair (f, g) ∈ V do do
8:              V = V − (f, g)
9:              Spoly(f, g) →G r
10:             if r ≠ 0 then
11:                 G = G ∪ r
12:                 V = V ∪ (G × r)
        return G
```

Example 1. Let $f = 6*{x_1}^4*{x_2}^5+24*{x_1}^2-x_2$ and $g = 2*{x_1}^2*{x_2}^7+4*{x_2}^3+2*x_3$ and we have $x_1 > x_2 > x_3$. The S-polynomial of f and g is defined below:

$$\mathrm{LM}(f) = {x_1}^4 * {x_2}^5$$

$$\mathrm{LM}(g) = {x_1}^2 * {x_2}^7$$

$$\mathrm{LCM}({x_1}^4 * {x_2}^5, {x_1}^2 * {x_2}^7) = {x_1}^4 * {x_2}^7$$

$$\begin{aligned}\mathrm{Spoly}(f,g) &= \frac{{x_1}^4 * {x_2}^7}{6 * {x_1}^4 * {x_2}^5} * f - \frac{{x_1}^4 * {x_2}^7}{2 * {x_1}^2 * {x_2}^7} * g \\ &= 4{x_1}^2 * {x_2}^2 - \frac{1}{6} * {x_2}^3 - 2 * {x_1}^2 * {x_2}^3 - {x_1}^2 * x_3\end{aligned}$$

□

It is obvious that S-polynomial computation cancels leading terms of the polynomials. As shown in Algorithm 9.1, Buchberger's algorithm first calculates all S-polynomials (lines 7-9 of Algorithm 9.1) and then adds non-zero S-polynomials to the basis G (line 11). This process repeats until all of the computed S-polynomials become zero with respect to G. It is obvious that Gröebner basis can be extremely large so its computation may take a long time and it may need large storage memory as well. The time and space complexity of this algorithm are exponential in terms of the sum of the total degree of polynomials in F, plus the sum of the lengths of the polynomials in F [4]. When the size of F increases, the verification process may be very slow or in the worst-case may be infeasible.

Buchberger's algorithm is computationally intensive and it may affect the performance drastically. It has been shown in [5] that if every pair (f_i, f_j) that belongs to set $F = \{f_1, f_2, \ldots, f_s\}$ (generator of ideal I) has a relatively prime leading monomials ($\mathrm{lm}(f_i).\mathrm{lm}(f_j) = \mathrm{LCM}(\mathrm{lm}(f_i).\mathrm{lm}(f_j))$) with respect to order $>$, the set F is also Gröbner basis of ideal I.

Based on these observations, efficient equivalence checking between specification of an arithmetic circuit and its implementation can be performed as shown in Fig. 9.4. The major computation steps in Fig. 9.4 are outlined below:

- Assuming a computational field $\mathbb{K}$ and a polynomial ring $\mathbb{K}[x_1, x_2, \ldots, x_n]$ (note that variables $\{x_1, x_2, \ldots, x_n\}$ are subset of signals in the gate-level implementation), a polynomial $f_{\mathrm{spec}} \in \mathbb{K}[x_1, x_2, \ldots, x_n]$ representing specification of the arithmetic circuit can be derived.
- Map the implementation of arithmetic circuit to a set of polynomials that belongs to $\mathbb{K}[x_1, x_2, \ldots, x_n]$. The set F generates an ideal I. Note that according to the field $\mathbb{K}$, some vanishing polynomials that construct ideal I_0 may be considered as well.
- Derive an order $>$ in a way that leading monomials of every pair (f_i, f_j) are relatively prime. Thus, the generator set F is also Gröbner basis $G = F$. As the combinational arithmetic circuits are acyclic, the topological order of the signals in the gate-level implementation can be used.

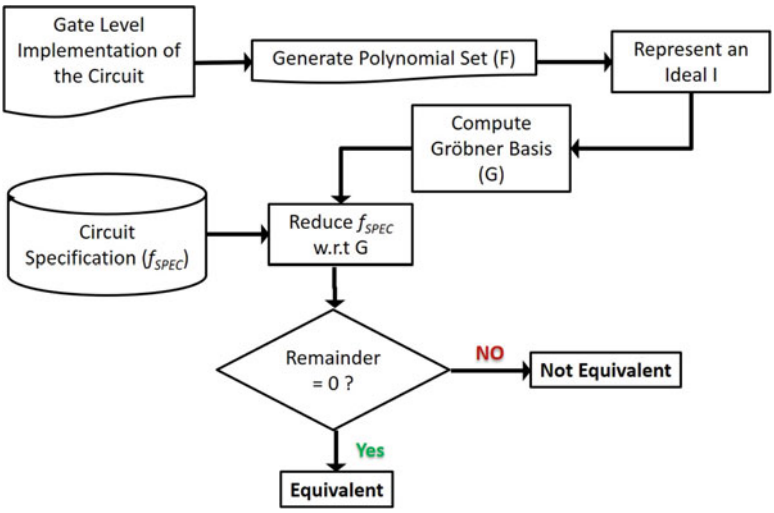


Fig. 9.4 Equivalence checking flow

- The final step is reduction of f_{spec} with respect to Gröbner basis G and order $>$. In other words, the verification problem is formulated as $f_{\text{spec}} \xrightarrow{G}_{+} r$. The gate-level circuit C has correctly implemented specification f_{spec}, if the remainder r is equal to 0. The non-zero remainder implies a bug or Trojan in the implementation.

Galois field arithmetic computation can be seen in Barrett reduction [16], Mastrovito multiplication, and Montgomery reduction [17] which are critical part of cryptosystems. In order to apply the method of Fig. 9.4 for verification of Galios field arithmetic circuits, Strong Nullstellensatz over Galois Fields is used. Galois field is not an algebraically closed field, so its closure should be used. Strong Nullstellensatz helps to construct a radical ideal in a way such that $I(V_{\mathbb{F}^{2^k}}) = I + I_0$. Ideal I_0 is constructed by using vanishing polynomials $x_i^{2^k} - x_i$ by considering the fact that $\forall x_i^{2^k} \in \mathbb{F}_{2^k} : x_i^{2^k} - x_i = 0$. As a result, the Gröbner basis theory can be applied on Galois field arithmetic circuits. The method in [18] has extracted circuit polynomials by converting each gate to a polynomial and SINGULAR [9] has been used to do the $f_{\text{spec}} \xrightarrow{G}_{+} r$ computations. Using this method, the verification of Galois field arithmetic circuits like Mastrovito multipliers with up to 163 bits can be done in few hours. Some extensions of this method have been proposed in [19]. The cost of $f_{\text{spec}} \xrightarrow{G}_{+} r$ computation has been improved by mapping the computation on a matrix representing the verification problem, and the computation is performed using Gaussian elimination.

The Gröbner basis theory has been used to verify arithmetic circuits over ring $\mathbb{Z}[x_1, x_2, \ldots, x_n]/2^N$ in [12]. Instead of mapping each gate to a polynomial, the

repetitive components of the circuit are extracted and the whole component is represented using one polynomial (since arithmetic circuit over ring $\mathbb{Z}[x_1, x_2, \ldots, x_n]/2^N$ contain carry chain, the number of polynomials can be very large). Therefore, the number of circuit polynomials is decreased. In order to expedite the $f_{\text{spec}} \xrightarrow{G}_{+} r$ computation, the polynomials are represented by Horner Expansion Diagrams. The reduction computation is implemented by sequential division. The verification of arithmetic circuit over ring $\mathbb{Z}[x_1, x_2, \ldots, x_n]/2^N$ up to 128 bit can be efficiently performed using this method. An extension of this method has been presented in [10] that is able to significantly reduce the number of polynomials by finding fanout free regions and representing the whole region by one single polynomial. Similar to [19], the reduction of specification polynomial with respect to Gröbner basis polynomials is performed by Gaussian elimination resulting in verification time of few minutes. In all of these methods, when the remainder r is non-zero, it shows that the specification is not exactly equivalent with the gate-level implementation. Thus, the non-zero remainder can be analyzed to identify the hidden malfunctions or Trojans in the system. In this section, the use of one of these approaches for equivalence checking of integer arithmetic circuits over $\mathbb{Z}_{2^n}$ is explained. Although the details are different for Galios Field arithmetic circuits, the major steps are similar.

9.3.2 *Automated Debugging of Functional Trojans Using Remainders*

The previous section describes that a non-zero remainder would be generated if there is a potential Trojan or bug. This section describes how to debug a Trojan using the non-zero remainder. To perform verification, the algebraic model of the implementation is used. In other words, each gate in the implementation is modeled as a polynomial with integer coefficients and variables from $\mathbb{Z}_2$ ($x \in \mathbb{Z}_2 \rightarrow x^2 = x$). Variables can be selected from primary inputs/outputs as well as internal signals in the implementation. These polynomials are driven in a way that they describe the functionality of a logic gate. Equation (9.1) shows the corresponding polynomial of *NOT, AND, OR, XOR* gates. Note that any complex gate can be modeled as a combination of these gates and its polynomial can be computed by combining the equations shown in Eq. (9.1).

$$\begin{aligned} z_1 &= NOT(a) \rightarrow z_1 = 1 - a, \\ z_2 &= AND(a, b) \rightarrow z_2 = a.b, \\ z_3 &= OR(a, b) \rightarrow z_3 = a + b - a.b, \\ z_4 &= XOR(a, b) \rightarrow z_4 = a + b - 2.a.b \end{aligned} \tag{9.1}$$

Finding potential Trojans starts with functional verification. The verification method is based on transforming specification polynomial (f_{spec}) using information (polynomials) that are directly extracted from the gate-level implementation. Then, the transformed specification polynomial is checked to see if the equality to zero holds. To fulfill the term substitution, the topological order of the circuit is considered (primary outputs have the highest order and primary inputs have the lowest). By considering the derived variable ordering, each non-primary input variable which exists in the f_{spec} is replaced with its equivalent expression based on corresponding implementation polynomial. Then, the f_{spec_i} is achieved and the process is continued on the updated $f_{\text{spec}_{i+1}}$ until a zero polynomial or a polynomial that only contains primary inputs (remainder) is reached. Note that, using a fixed variable (term) ordering to substitute the terms in the f_{spec_i}s results in having a unique remainder [8]. The following example shows the verification process of a faulty 2-bit multiplier.

Example 2. Suppose, we want to verify a 2-bit multiplier with gate-level netlist shown in Fig. 9.5 to check there is no additional functionality beside the main functionality in the implementation. Suppose a functional hardware Trojan exists in the design by putting the OR gate with inputs (A_1, B_0) instead of an AND gate as in Fig. 9.5. The specification of a 2-bit multiplier is shown by f_{spec_0}. The verification process starts from f_{spec_0} and replaces its terms one by one using information derived from implementation polynomials as shown in Eq. (9.2). For instance, term $4.Z_2$ from f_{spec_0} is replaced with expression $(R + O - 2.R.O)$. The topological order $\{Z_3, Z_2\} > \{Z_1, R\} > \{Z_0, M, N, O\} > \{A_0, A_1, B_0, B_1\}$ is considered to perform term rewriting. The verification result is shown in Eq. (9.2). Clearly, the remainder is a non-zero polynomial and it reveals the fact that the implementation is buggy. □

$$
\begin{aligned}
&f_{\text{spec}_0} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&f_{\text{spec}_1} : 4.R + 4.O + 2.z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&f_{\text{spec}_2} : 4.O + 2.M + 2.N + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&f_{\text{spec}_3}(\text{remainder}) : 2.A_1 + 2.B_0 - 4.A_1.B_0
\end{aligned}
\tag{9.2}
$$

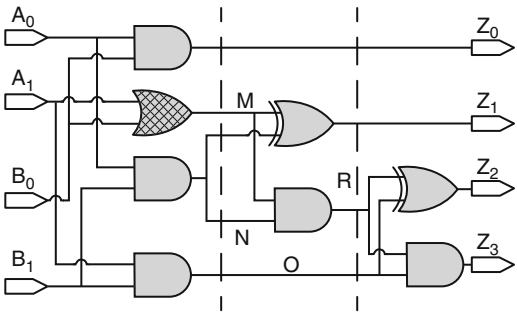


Fig. 9.5 A Trojan-inserted gate-level netlist of a 2-bit multiplier (an AND gate is erroneous and it is replaced by the shaded OR gate)

9.3.2.1 Test Generation for Trojan Detection

It has been shown that if the remainder is zero, there is no malicious functionality in the implementation and implementation is bug free [30]. Thus, when there is a non-zero polynomial as a remainder, any assignment to its variables that makes the decimal value of the remainder non-zero is a bug trace. The remainder is used to generate test cases to activate unknown faults or inserted Trojans [11]. The test is guaranteed to activate the existing malicious functionality in the design. Remainder is a polynomial with Boolean/integer coefficients. It contains a subset of primary inputs as its variables. The presented approach in [11] takes the remainder as an input and finds all of the assignments to its variables such that it makes the decimal value of the remainder non-zero. The remainder may not contain all of the primary inputs. As a result, the approach may use a subset of primary inputs (that appear in the remainder) to generate directed test cases with *don't cares*.

Such assignments can be found using an SMT solver by defining Boolean variables and considering signed/unsigned integer values as total value of the remainder polynomial ($i \neq 0 \in \mathbb{Z}, \text{check}(R = i)$). The problem of using SMT solver is that for each i, it finds at most one assignment to the remainder variables to produce value of i, if possible.

Algorithm 9.2 finds all possible assignments which produce non-zero decimal values of the remainder. It gets remainder R polynomial and primary inputs (PI) exists in the remainder as inputs and feeds binary values to PIs (s_i) and computes the total value of a term (T_j). The value of T_j is either one or zero as it is multiplication of some binary variables (line 4–5). The whole term value may be zero or equal to the term coefficient (C_{T_j}). Then, it computes sum of all terms value to find the corresponding value of the remainder polynomial. If the summation (value) of all the terms is non-zero, the corresponding primary input assignments are added to the set of Tests (lines 8–9).

Example 3. Consider the faulty circuit shown in Fig. 9.5 and the remainder polynomial $R = 2.(A_1 + B_0 - 2.A_1.B_0)$. The only assignments that make R to have a non-zero decimal value ($R = 2$) are ($A_1 = 1, B_0 = 0$) and ($A_1 = 0, B_0 = 1$).

Algorithm 9.2: Directed Test Generation Algorithm

```
1: procedure TEST–GENERATION
2:     Input: Remainder, R
3:     Output: Directed Tests, Tests
4:     for different assignments s_i of PIs in R do
5:         for each term T_j ∈ R do
6:             if (T_j(s_i)) then
7:                 Sum+ = C_Tj
8:         if ( Sum != 0 ) then
9:             Tests = Tests ∪ s_i
       return Tests
```

Table 9.1 Directed tests to activate functional Trojan shown in Fig. 9.5

A_1	A_0	B_1	B_0
1	X	X	0
0	X	X	1

These are the only scenarios that make difference between functionality of an AND gate and an OR gate. Otherwise, the fault will be masked. Compact directed test cases are shown in Table 9.1. □

The remainder generation is one time effort, multiple directed tests can be generated by using it. Moreover, if there are more than one bug in the implementation, the remainder will be affected by all of the bugs. So, each assignment that makes the remainder non-zero activates at least one of the existing faulty scenarios. Thus, the proposed test generation method can also be applied when there are more than one fault in the design.

Example 4. Suppose that in circuit shown in Fig. 9.5, the AND gate with inputs (A_0, B_1) is also replaced with an OR gate by mistake (therefore, there are two functional Trojans in the implementation of the 2 bit multiplier). The remainder polynomial will be equal to $R = 6.A_1 + 4.B_1 + 2.B_0 - 12.A_1.B_1 - 4.A_1.B_0$. It can be seen that assignment $(A_1 = 1, A_0 = 0, B_1 = 0, B_0 = 0)$ manifests the effect of the first fault in Z_1 and assignment $(A_1 = 1, A_0 = 0, B_0 = 0, B_0 = 1)$ activates the second fault in Z_2. □

9.3.2.2 Trojan Localization

So far, we know that the implementation is not correct and we have all the necessary test cases to activate the malicious scenarios. Next goal is to reduce the state space in order to localize the Trojan by using test cases generated in the previous section. The Trojan location can be traced by observing the fact that the outputs can possibly be affected by the existing Trojan. The proposed methodology is based on simulation of test cases.

The tests are simulated and the outputs are compared with the golden outputs (golden outputs can be found from the specification polynomials) to track the faulty (unmatched) outputs in set $E = \{e_1, e_2, .., e_n\}$. Each e_i denotes one of the erroneous outputs. To localize the bug/hardware Trojan, the gate-level netlist is partitioned such that fanout free cones (set of gates that are directly connected together) of the implementation are found. Algorithm 9.3 shows the procedure of partitioning of gate-level netlist of a circuit.

Each gate whose output goes to more than one gate is selected as a fanout. For generality, gates that produce primary outputs are also considered as fanouts. Algorithm 9.3 takes gate-level netlist (*Imp*) and fanout list (L_{fo}) of a circuit as inputs and returns fanout free cones as its output. Algorithm 9.3 chooses one fanout gate from L_{fo} and goes backward from that fanout until it reaches the gate g_i, whose input comes from one of the fanouts from L_{fo} or primary inputs; the algorithm marks all the visited gates as a cone.

Algorithm 9.3: Fanout free cone finder algorithm

```
1: procedure FANOUT–FREE–CONE–FINDER
2:     for Each fanout gate g_i ∈ L_fo do
3:         C.add(g)
4:         for All inputs g_j of g_i do
5:             if !(g_j ∈ L_fo ∪ PI) then
6:                 C.add(g_j)
7:                 Call recursive for all inputs of g_j over Imp
8:                 Add found gates to C
9:         Cones = Cones ∪ C
```

Algorithm 9.4: Bug Localization Algorithm

```
1: procedure BUG–LOCALIZATION
2:     Input: Partitioned Netlist, Faulty Outputs E
3:     Output: Suspected Regions C_S
4:     for each faulty output e_i ∈ E do
5:         find cones that construct e_i and put in C_ei
6:     C_S = C_e0
7:     for e_i ∈ E do
8:         C_S = C_S ∩ C_ei
       return C_S
```

Algorithm 9.4 shows the bug/Trojan localization procedure. Given a partitioned erroneous circuit and a set of faulty (unmatched) outputs E, the goal of the automatic bug localization is to identify all of the potentially responsible cones for the malicious functionality. First, sets of cones $C_{e_i} = \{c_1, c_2, \ldots, c_j\}$ that construct the value of each e_i from set E are found (line 4–5). These cones contain suspicious gates. All of the suspicious cones C_{e_i}s are intersected to prune the search space and improve the efficiency of Trojan localization algorithm. The intersection of these cones are stored in C_S (line 7–8).

If simulating all of the tests show the effect of the malicious behavior in just one of the outputs, it can be concluded that the location of the bug is in the cone that generates this output. Otherwise, the effect of bug might have been detected in other outputs. On the other hand, when the effect of the bug can be observed in all of the outputs, it means that the bug location is closer to the primary inputs as the error has been propagated through all the circuit. Thus, the location of the bug is in the intersection of cones which constructs the faulty outputs.

Example 5. Consider the faulty 2-bit multiplier shown in Fig. 9.6. Suppose the AND gate with inputs (M, N) has been replaced with an OR gate by mistake. So, the remainder is $R = 4.A_1.B_0 + 4.A_0.B_1 - 8.A_0.A_1.B_0.B_1$. The assignments that activate the Trojan are calculated based on Algorithm 9.2. Tests are simulated and the faulty outputs are obtained as $E = \{Z_2, Z_3\}$. Then, the netlist is partitioned to find fanout free cones. The cones involved in construction of faulty outputs are: $C_{Z_2} = \{2, 3, 4, 6, 7\}$ and $C_{Z_3} = \{2, 3, 6, 4, 8\}$. The intersection of the cones that produce faulty outputs is $C_S = \{2, 3, 4, 6\}$. As a result, gates $\{2, 3, 4, 6\}$ are potentially responsible as a source of the error. □

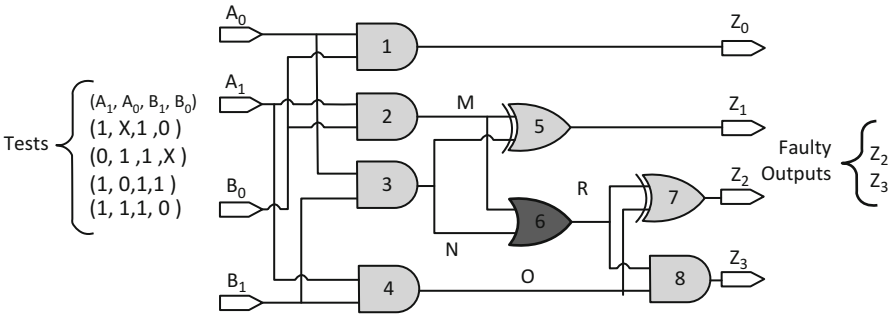


Fig. 9.6 A Trojan-inserted gate-level netlist of a 2-bit multiplier with associated tests to activate the Trojan

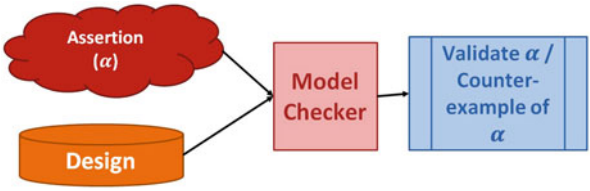


Fig. 9.7 Model checker functionality

9.4 Trojan Detection Using Model Checking

A model checker checks a design against its specification properties (assertions). It takes the design along with the properties (functional behaviors written as Linear Time Temporal Logic formulas) and it formally checks whether properties are satisfied for all of possible states of the design implementation. The high level overview of a model checker functionality is shown in Fig. 9.7. Design specification can be formulated as a set of Linear Time Temporal Logic (LTL) properties [21]. Each property describes one expected behavior of the design. Example 6 shows a property related to a router design.

Example 6. Figure 9.8 shows a router that receives a packet of data from its input channel. The router analyzes the received packet and sends it to one of the three channels based on the packet's address. F_1, F_2, and F_3 receive packets with address of 1, 2, and 3, respectively. Input data consists of three parts: (1) parity (*data_in*[0]) (2) payload (*data_in*[7..3]), and (3) address (*data_in*[2..1]). The RTL implementation consists of one FIFO connected to its input port (F_0) and three FIFOs (one for each of the output channels). The FIFOs are controlled by an FSM. The routing module reads the input packet and asserts the corresponding target FIFO's write signal (write1, write2, and write3).

Suppose the designer wants to make sure that whenever router receives a packet with address 1 (*data_in*[2..1] $= 2'b01$), it eventually resides in the F_1 fifo which is responsible for receiving the data with address 1 (the corresponding write signal of the F_1 fifo will be asserted). The property can be formulated as an LTL formula as follows:

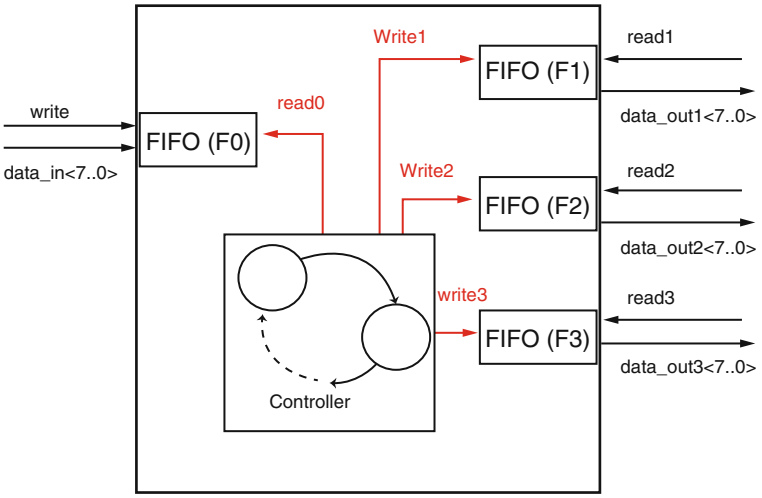


Fig. 9.8 Block diagram of a router design

assert $P1$: *always* $((F_0.data\_in[2] = 0 \wedge F_0.data\_in[1] = 1) \rightarrow$ *eventually* $(write1 = 1))$ □

To formally verify the design, model checkers check the design for all possible states. For example, if a design has a 5-bit input, the input introduces 2^5 different states for the design. Therefore, model checkers have state-space constraint when they are applied on large and complex design. Bounded model checking is used to overcome some limitations of model checkers. It checks the design for certain number of transitions and if it finds a violation, it produces a counter-example for the target property [2]. A bounded model checker unrolls the circuit for certain number of clock cycles and extracts Boolean satisfiability assignments for the unrolled design. It feed Boolean satisfiability assignments to the SAT solver to search for assignments that cause the assertion to fail. The model checker generates a counter-example whenever it faces a violation in the implementation of a property. The high level overview of a bounded model checker is shown in Fig. 9.9. Since bounded model checker does not check all states, it cannot formally validate the given property. In bounded model checking, the assumption is that the designer knows in how many cycles a property should be held. Then, the number of cycles (N) is fed to SAT solver engine to find a counter-example within N state sequence transitions (within N clock cycles).

Karri et al. [22] proposed a method to use bounded model checkers to detect malicious functionality in the design. Their considered threat model is that whether an attacker can corrupt the data residing in N-bit data critical registers such as registers containing secret keys of a cryptography design, stack pointer of a processor, or address of a router. A set of properties is written describing that a critical register's data should be accessed safely. Then properties are fed into

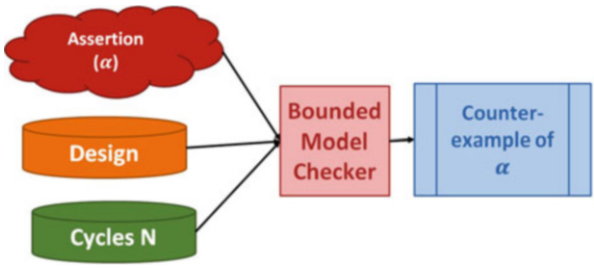


Fig. 9.9 Bounded model checker functionality

bounded model checker accompanied with the design. The bounded checker tires to find a set of states showing unauthorized access to the critical registers which violates the given property.

For example, it is important to make sure that the stack pointer of a processor is only accessed by one of these three ways: (1) increment by one instruction (I_1); (2) decrement by one instruction (I_2); (3) using the reset signal (I_3). The stack pointer should be only accessed from three mentioned ways and if it is accessed from a different way, it should be considered as a malfunction and a security threat. The property can be written as follows:

$$PC\_safe\_access : assert \quad \text{always} \quad (I_{\text{trigger}} \notin I = \{I_1, I_2, I_3\} \rightarrow PC_t = PC_{t-1})$$

The property is fed to a bounded model checker and malfunctions are detected whenever the model checker finds a counter-example for the given property. However, this approach requires prior knowledge to all of the safe ways to access critical data which may not be available. Moreover, translation of safe access ways to LTL formulas is not straightforward. Model checkers fail for large and complex designs and they need the design in a specific language/format. Translation of RTL design to those languages can be error-probe. Moreover, it will face state-space explosion when the Trojan is triggered after thousands or millions of cycles.

Model checkers and theorem provers are combined together in [14] to address the scalability and overhead issues of both methods in verifying security properties. In this method, security properties are translated to formal theorems. In the next step, theorems are converted into disjoint lemmas with respect to the design's submodules. Next, lemmas are modeled as Property Specification Language (PSL) assertions. Then model checker is invoked to check the presence of Trojans in the design.

References

1. M. Banga, M. Hsiao, Trusted RTL: trojan detection methodology in pre-silicon designs, in *Proceedings of 2010 IEEE Internationl Symposium on Hardware-Oriented Security and Trust (HOST)* (2010), pp. 56–59

2. A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without bdds, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Springer, Berlin, 1999), pp. 193–207
3. E. Biham, Y. Carmeli, A. Shamir, Bug attacks, in *CRYPTO 2008*, ed. by Wagner, D. LNCS, vol. 5157 (Springer, Heidelberg, 2008), pp. 221–240
4. B. Buchberger, Some properties of gröbner-bases for polynomial ideals. ACM SIGSAM Bull. **10**(4), 19–24 (1976)
5. B. Buchberger, A criterion for detecting unnecessary reductions in the construction of a groebner bases, in *EUROSAM* (1979)
6. R.S. Chakraborty, F. Wolf, C. Papachristou, S. Bhunia, Mero: a statistical approach for hardware trojan detection, in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES'09)* (2009), pp. 369–410
7. M.J. Ciesielski, C. Yu, W. Brown, D. Liu, A. Rossi, Verification of gate-level arithmetic circuits by function extraction, in *IEEE/ACM International Conference on Computer Design Automation (DAC)* (2015), pp. 1–6
8. D. Cox, J. Little, D. O'shea, Ideal, Varieties and Algorithm: An Introduction to Computational Algebraic Geometry and Commutative Algebra (Springer, Berlin, 2007)
9. W. Decker, G.-M. Greuel, G. Pfister, H. Schönemann, SINGULAR 3.1.3 A Computer Algebra System for Polynomial Computations. Centre for Computer Algebra (2012). http://www.singular.uni-kl.de
10. F. Farahmandi, B. Alizadeh, Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction, in *Microprocessors and Microsystems - Embedded Hardware Design* (2015), pp. 83–96
11. F. Farahmandi, P. Mishra, Automated test generation for debugging arithmetic circuits, in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE, New York, 2016), pp. 1351–1356
12. F. Farahmandi, B. Alizadeh, Z. Navabi, Effective combination of algebraic techniques and decision diagrams to formally verify large arithmetic circuits, in *2014 IEEE Computer Society Annual Symposium on VLSI* (IEEE, New York, 2014), pp. 338–343
13. X. Guo, R.G. Dutta, Y. Jin, F. Farahmandi, P. Mishra, Pre-silicon security verification and validation: a formal perspective, in *ACM/IEEE Design Automation Conference (DAC)* (2015)
14. X. Guo, R.G. Dutta, P. Mishra, Y. Jin, Scalable soc trust verification using integrated theorem proving and model checking, in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2016)
15. M. Hicks, M. Finnicum, S. King, M. Martin, J. Smith, Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *IEEE Symposium on Security and Privacy (SP)* (2010), pp. 159–172
16. M. Knežević, K. Sakiyama, J. Fan, I. Verbauwhede, Modular reduction in gf (2 n) without pre-computational phase, in *International Workshop on the Arithmetic of Finite Fields* (Springer, Berlin, 2008), pp. 77–87
17. C. Koc, T. Acar, Montgomery multiplication in $GF(2^k)$, Des. Codes Crypt. **14**(1), 57–69 (1998)
18. J. Lv, P. Kalla, F. Enescu, Efficient groebner basis reductions for formal verification of galois field multipliers, in *Proceedings Design, Automation and Test in Europe Conference (DATE)* (2012), pp. 899–904
19. J. Lv, P. Kalla, F. Enescu, Efficient groebner basis reductions for formal verification of galois field arithmetic circuits. IEEE Trans. CAD **32**, 1409–1420 (2013)
20. M. Oya, Y. Shi, M. Yanagisawa, N. Togawa, A score-based classification method for identifying hardware-trojans at gate-level netlists, in *Design Automation and Test in Europe (DATE)* (2015), pp. 465–470
21. A. Pnueli, The temporal semantics of concurrent programs, in *Semantics of Concurrent Computation* (Springer, Berlin, 1979), pp. 1–20
22. J. Rajendran, V. Vedula, R. Karri, Detecting malicious modifications of data in third-party intellectual property cores, in *Proceedings of the 52nd Annual Design Automation Conference* (ACM, New York, 2015), p. 112

23. S. Saha, R. Chakraborty, S. Nuthakki, Anshul, D. Mukhopadhyay, Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability, in *Cryptographic Hardware and Embedded Systems (CHES)* (2015), pp. 577–596
24. N. Shekhar, P. Kalla, F. Enescu, Equivalence verification of polynomial datapaths using ideal membership testing. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **26**(7), 1320–1330 (2007)
25. C. Sturton, M. Hicks, D. Wagner, S. King, Defeating uci: building stealthy and malicious hardware, in *IEEE Symposium on Security and Privacy (SP)* (2011), pp. 64–77
26. X. Sun, P. Kalla, T. Pruss, F. Enescu, Formal verification of sequential galois field arithmetic circuits using algebraic geometry, in *Design Automation and Test in Europe (DATE)* (2015), pp. 1623–1628
27. A. Waksman, S. Sethumadhavan, Silencing hardware backdoors, in *2011 IEEE Symposium on Security and Privacy* (IEEE, New York, 2011), pp. 49–63
28. A. Waksman, M. Suozzo, S. Sethumadhavan, Fanci: identification of stealthy malicious logic using boolean functional analysis, in *ACM SIGSAC Conference on Computer & Communications Security* (2013), pp. 697–708
29. A. Waksman, S. Sethumadhavan, J. Eum, Practical, lightweight secure inclusion of third-party intellectual property. IEEE Des. Test Comput. **30**(2), 8–16 (2013)
30. O. Wienand, M. Welder, D. Stoffel, W. Kunz, G.M. Greuel, An algebraic approach for proving data correctness in arithmetic data paths, in *Computer Aided Verification (CAV)* (2008), pp. 473–486
31. X. Zhang, M. Tehranipoor, Case study: detecting hardware trojans in third-party digital ip cores, in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (IEEE, New York, 2011), pp. 67–70
32. J. Zhang, F. Yuan, Q. Xu, Detrust: defeating hardware trust verification with stealthy implicitly-triggered hardware trojans, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (ACM, New York, 2014), pp. 153–166
33. J. Zhang, F. Yuan, L. Wei, Y. Liu, Q. Xu, Veritrust: verification for hardware trust. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(7), 1148–1161 (2015)

Chapter 10
IP Trust Validation Using Proof-Carrying Hardware

Xiaolong Guo, Raj Gautam Dutta, and Yier Jin

10.1 Introduction

A rapidly growing third-party Intellectual Property (IP) market provides IP consumers with high flexibility when designing electronic systems. It also reduces the development time and expertise needed to compete in a market where profit-windows are very narrow. However, one key issue that has been neglected is the security of hardware designs built upon third-party IP cores. Historically, IP consumers have focused on IP functionality and performance than security. The negligence toward development of robust security policies is reflected in the IP design flow (see Fig. 10.1), where IP core specification usually only includes functionality and performance measurements.

The prevailing usage of third-party soft IP cores in SoC designs raises security concerns as current IP core verification methods focus on IP functionality rather than IP trustworthiness. Moreover, lack of regulation in the IP transaction market adds to the predicament of the SoC designers and forces them to perform verification and validation of IPs themselves. To help SoC designers in IP verification, various methods have been developed to leverage enhanced functional testing and/or perform probability analysis of internal nodes for IP core trust evaluation and malicious logic detection [1, 2]. However, these methods were easily bypassed by sophisticated hardware Trojans [3–5]. Formal methods were also introduced for IP core trust evaluation [1, 6–10]. Among all the proposed formal methods, proof-carrying hardware (PCH), which originated from proof-carrying code (PCC), emerged as one of the most prevalent methods for certifying the absence of malicious logic in soft IP cores and reconfigurable logic [6–10]. In the PCH approach, synthesizable register-transfer level (RTL) code of IP core and informal security properties were

X. Guo • R.G. Dutta • Y. Jin (✉)
University of Central Florida, Orlando, FL 32816, USA
e-mail: guoxiaolong@knights.ucf.edu; rajgautamdutta@knights.ucf.edu; yier.jin@eecs.ucf.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_10

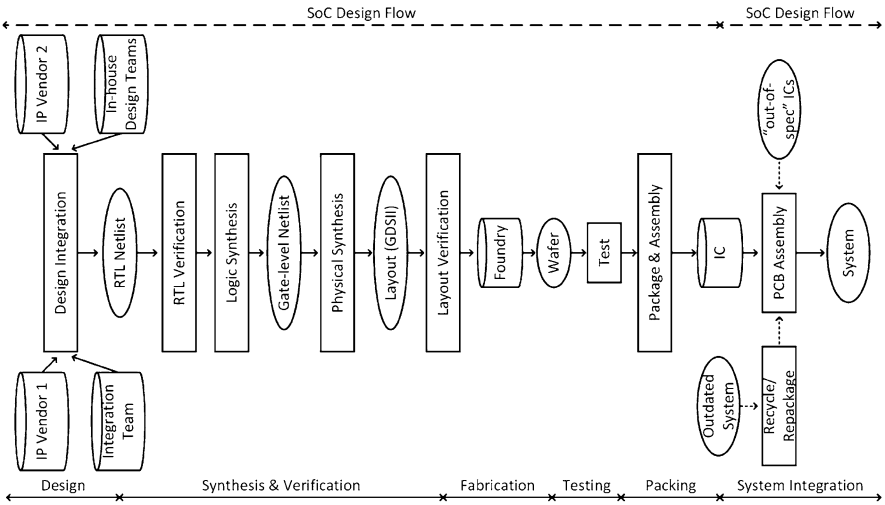


Fig. 10.1 IC design flow within the semiconductor supply chain

first represented in *Gallina*—the internal functional programming language of the Coq proof assistant [11]. Then, Hoare-logic style reasoning was used to prove the correctness of the RTL code in the Coq platform.

The rest of the chapter is organized as follows: In Sect. 10.2, we provide an overview of existing methods for IP protection, introduce the threat model, and provide some relevant background on two different formal verification approaches. In Sect. 10.3, we provide detailed explanation of the PCH method for ensuring trustworthiness of IP cores. Finally, Sect. 10.4 concludes the chapter.

10.2 Overview of Formal Verification Methods for IP Protection

To counter the threat of untrusted third-party resources, pre-silicon trust evaluation approaches have been proposed recently [1, 12, 13]. Most of these methods try to trigger malicious logic by enhancing functional testing with extra test vectors. Authors in [12] proposed a method to generate "Trojan Vectors" into the testing patterns, hoping to activate the hardware Trojans during the functional testing. In order to identify suspicious circuitry, unused circuit identification (UCI) [13] method analyzed the RTL code to find lines of code that are never used. However, these methods assume that the attacker uses rarely occurring events as Trojan triggers. Using "less-rare" events as trigger will void these approaches. This was demonstrated in [14], where hardware Trojans were designed to defeat UCI.

Admitting the limitations of enhanced functional testing methods, researchers started looking into formal solutions. Although at its early stage, formal methods have already shown their advantages over testing methods in exhaustive security verification [8, 9, 15, 16]. A multi-stage approach, which included assertion based verification, code coverage analysis, redundant circuit removal, equivalence analysis, and use of sequential Automatic Test Pattern Generation (ATPG), was adopted in [15] to identify suspicious signals for detecting hardware Trojans. This approach was demonstrated on an RS232 circuit and the efficiency of the approach in detecting Trojan signals ranged between 67.7 and 100 %. In [8, 9, 16], a PCH framework was used to verify security properties on soft IP cores. Supported by the Coq proof assistant [11], formal security properties were formalized and proved to ensure the trustworthiness of IP cores. In the following section in this chapter, we will explain the PCH approach for soft IP core verification in greater details. This method uses an interactive theorem prover and model checker for verifying the design.

10.2.1 Threat Model

The IP protection methods in this chapter are based on the threat model that malicious logic is inserted by an adversary at the design stage of the supply chain. We assume that the rogue agent at the third-party IP design house can access the hardware description language (HDL) code and insert a hardware Trojan or backdoor to manipulate critical registers of the design. Such a Trojan can be triggered either by a counter at a predetermined time, by an input vector, or under certain physical conditions. Upon activation it can leak sensitive information from the chip, modify functionality, or cause a denial-of-service to the hardware. In this chapter, Trojans which can be activated by a specific "digital" input vector are only considered.

Meanwhile, verification tools (e.g., Coq) used in all methods are assumed to produce correct results. The existence of proofs for the security theorems indicates the genuineness of the design whereas its absence indicates the presence of malicious logic. However, the framework does not provide protection of an IP from Trojans whose behaviors are not captured by the set of security properties. Furthermore, there is also an assumption that the attacker has intricate knowledge of the hardware to identify critical registers and modify them in order to carry out the attack.

10.2.2 Formal Verification Methods

Formal methods have been extensively used for verification and validation of security properties at pre- and post-silicon stages [8, 9, 15–20]. These previous methods leverage one of the following two techniques, model checking and interactive/automated theorem proving, for design verification.

10.2.2.1 Theorem Prover

Theorem provers are used to prove or disprove properties of systems expressed as logical statements [21–28]. Over the years, several theorem provers (both interactive and automated) have been developed for proving properties of hardware and software systems. However, using them for verification on large and complex systems require excessive effort and time. Irrespective of these limitations, theorem provers have currently drawn a lot of interest in verifying security properties on hardware. Among all the formal methods, they have emerged as the most prominent solution for verifying large-scale designs.

One leading example of an interactive theorem prover is the open source tool called Coq proof assistant [11]. Coq is an interactive theorem prover/proof assistant, which enables verification of software and hardware programs with respect to their specification [25]. In Coq, programs, properties, and proofs are represented as terms in the *Gallina* specification language. By using the *Curry–Howard Isomorphism*, the interactive theorem prover formalizes both the program and proofs in its dependently typed language called the *Calculus of Inductive Construction (CIC)*. Correctness of the proof of the program is automatically checked using the built-in type-checker of Coq. To expedite the process of building proofs, Coq provides a library consisting of programs called *tactics*. However, existing Coq *tactics* does not capture properties of hardware designs and thus does not significantly reduce the time required for certifying large-scale hardware IP cores [6–8].

10.2.2.2 Model Checker

Model checking [29] is an automated method for verifying and validating models in software and hardware applications [30–42]. In this approach, a model (Verilog/VHDL code of hardware) $\mathscr{M}$ with an initial state s_0 is expressed as a transition system and its behavioral specification (assertion) ϕ is represented in a temporal logic. The underlying algorithm of this technique explores the state space of the model to find whether the specification is satisfied. This can be formally stated as, $\mathscr{M}, s_0 \models \phi$. If a case exists where the model does not satisfy the specification, a counterexample in the form of a trace is produced by the model checker [43, 44]. Recently, model checkers have been used for detecting malicious signals in third-party IP cores [15, 20]. The application of model checking techniques to SoCs, including symbolic approaches based on Reduced Order Binary Decision Diagrams (ROBDD) and Satisfiability (SAT) solving, has had limited success due to the state-space explosion problem [45]. For example, a model with n Boolean variables can have as many as 2^n states, a typical soft IP core with 1000 32-bit integer variables has billions of states.

Symbolic model checking using ROBDD is one of the initial approaches used for hardware systems verification [46–48]. Unlike explicit state model checking where all states of the system are explicitly enumerated, this technique model states (represented symbolically) of the transition system using ROBDD.

The ROBDD is a unique, canonical representation of a Boolean expression of the system. Subsequently, the specification to be checked is represented using a temporal logic. A model checking algorithm then checks whether the specification is true on a set of states of the system. Despite being a popular data structure for symbolic representation of states of the system, ROBDD requires finding an optimal ordering of state variables which is an NP-hard problem. Without proper ordering, the size of the ROBDD increases significantly. Moreover, it is memory intensive for storing and manipulating Binary Decision Diagrams (BDDs) of a system with a large state space.

Another technique called bounded-model checking (BMC) replaces BDDs in symbolic checking with SAT solving [49–51]. In this approach, a propositional formula is first constructed using a model of the system, the temporal logic specification, and a bound. The formula is then provided to an SAT solver to either obtain a satisfying assignment or to prove that no such assignment exists. Although BMC outperforms BDD based model checking in some cases, the method cannot be used to test properties (specification) when the bound is large or cannot be determined.

10.3 Proof-Carrying Hardware Framework for IP Protection

Various methods have been proposed in the software domain to validate the trustworthiness and genuineness of software programs. These methods protect computer systems from untrusted software programs. Most of these methods lay burden on software consumers to verify the code. However, *proof-carrying code* (PCC) switches the verification burden to software providers (software vendors/developers). Figure 10.2 outlines the basic working process of the PCC framework.

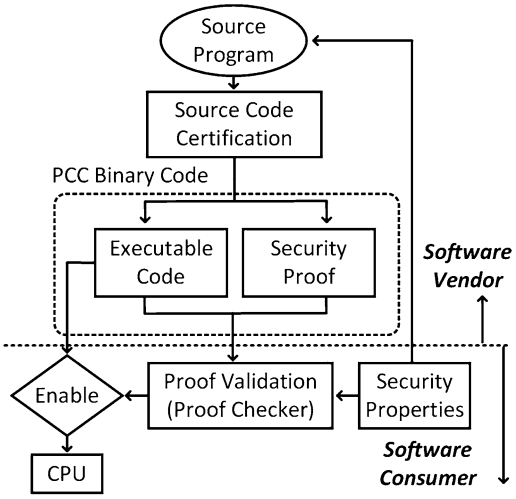


Fig. 10.2 Working procedure of the PCC framework [52]

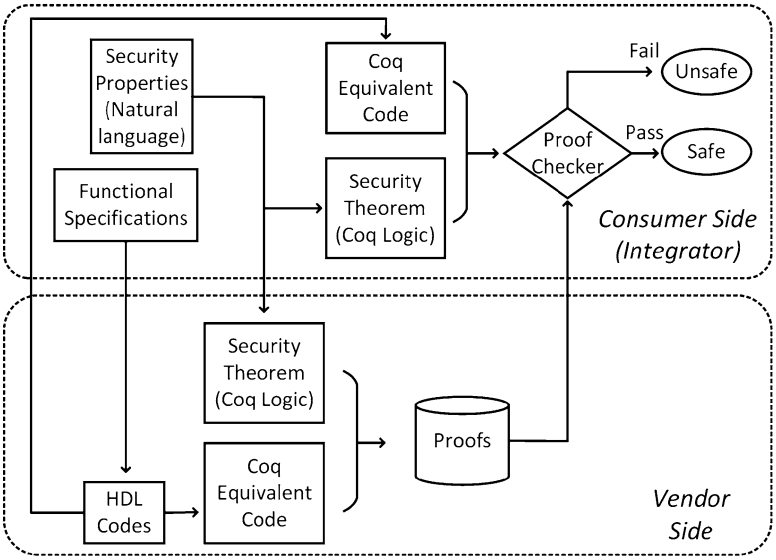


Fig. 10.3 Working process of the PCH framework [17]

During the *source code certification* stage of the PCC process, the software provider verifies the code with respect to the *security property* designed by the software consumer and encodes the formal proof of the security property with the executable code in a *PCC binary file*. In the *proof validation* stage, the software consumer determines whether the code from the potentially untrusted software provider is safe for execution by validating the PCC binary file using a proof checker [52].

A similar mechanism, referred to as Proof-Carrying Hardware (PCH), was used in the hardware domain to protect third-party soft IP cores [8–10]. The PCH framework ensures trust-worthiness of soft IP cores by verifying a set of carefully specified security properties. The working procedure of the PCH framework is shown in Fig. 10.3. In this approach, the IP consumer provides design specifications and informal (written in natural language) security properties to the IP vendor. Upon receiving the request, the IP vendor develops the RTL design using a hardware description language (HDL). Then, semantic translation of the HDL code and informal security properties to Gallina is carried out. Subsequently, Hoare-logic style reasoning is used for proving the correctness of the RTL code with respect to formally specified security properties in Coq. As Coq supports automatic proof checking, it can help IP customers validate proof of security properties with minimum efforts. Moreover, usage of the Coq platform by both IP vendors and IP consumers ensures that the same deductive rules will be used for validating the proof. After verification, the IP vendor provides the IP consumer with the HDL code (both original and translated versions), formalized security theorems of security properties, and proofs of these security theorems. Then, the proof checker in Coq is used by the IP consumer to quickly validate the proof of security theorems on the

translated code. The proof checking process is fast, automated, and does not require extensive computational resources.

10.3.1 Semantic Translation

The PCH based IP protection method requires semantic translation of circuit design in HDL to Coq's specification language, *Gallina*. Consequently, a *formal-HDL* is developed in [10], which includes a set of rules to enable this translation. These rules can help represent basic circuit units, combinational logic, sequential logic, and module instantiations. The *formal-HDL* is further extended in [53] to capture hierarchical design methodology, which is used for representing large circuits such as SoC. A brief description of the *formal-HDL* is given below

- *Basic Circuit Units:*
  In the *formal-HDL*, basic circuit units are the most important components and they include signals and buses. During the translation, three digital values are used for signals: high, low, and unknown. To represent sequential logic, a *bus* type is defined as a function, which takes timing variable *t* and returns a list of signal values as shown in *Listing 10.1*. All circuit signals are of *bus* type and their values can be modified either by a blocking assignment or a nonblocking assignment (shown in *Listing 10.1*). Moreover, inputs and outputs are also defined as *bus* type.

Listing 10.1 Basic Circuit Units in Semantic Model

```
Inductive value := lo|hi|x.
Definition bus_value := list value.
Definition bus := nat -> bus_value.
Definition input := bus.
Definition output := bus.
Definition wire := bus.
Definition reg := bus.
```

- *Signal Operations:* Logic operations such as *and*, *or*, *not*, and *xor*, as well as bus comparison operations such as checking for bus equality: *bus_eq* and less-than: *bus_lt* are designed to handle *bus* in *Gallina*. The conditional statement of RTL code such as *if*... *else*... checks whether signals are on or off. To incorporate this functionality in Coq, a special function, *bus_eq_0*, which compares the bus value to *hi* or *lo* is added.

Listing 10.2 Signal Operations in Semantic Model

```
Fixpoint bv_bit_and  (a b : bus_value) {struct a} : bus_value :=
  match a with
   | nil => nil
   | la :: a' =>
     match b with
      | nil => nil
      | lb :: b' => (v_and la lb)::(bv_bit_and a' b')
     end
  end.
Definition bus_bit_and (a b : bus) : bus :=
    fun t:nat => bv_bit_and (a t) (b t).
Fixpoint bv_eq_0 (a : bus_value) {struct a} : value :=
  match a with
   | hi :: lt => lo
   | lo :: lt => bv_eq_0 lt
   | nil => hi
  end.
Definition bus_eq_0 (a : bus) (t : nat) : value := bv_eq_0 (a t).
```

- *Combinational and Sequential Logic:* The definition of signals, expressions, and their semantics paves the way for converting RTL circuits into Coq representatives. Combinational and sequential logic are higher level logic descriptions constructed on top of buses. The keyword *assign* of the *formal-HDL* is used for blocking assignment, while *update* is mainly used for nonblocking assignment. During the blocking assignment the bus value will be updated in the current clock cycle and in the nonblocking assignment the bus value will be updated in the next clock cycle.

Listing 10.3 Signal Operations in Semantic Model

```
Fixpoint assign (a:assignblock)(t:nat) {struct a} :=
    (* Blocking assignment *)
match a with
| expr_assign bus_one e => bus_one t = eval e t
| assign_useless => True
| assign_cons a1 a2 => (assign a1 t) /\ (assign a2 t)
end.
Fixpoint update (u:updateblock)(t:nat) {struct u} :=
    (* Nonblocking assignment *)
match u with
| (upd_expr bus exp) => (bus (S t)) = (eval exp t)
| (updcons block1 block2) => (update block1 t) /\ (update block2 t)
| upd_useless => True
end.
```

- *Module Definitions:* Module definition/instantiation is critical when dealing with hierarchical circuit structures, but it is never a problem for Verilog (and VHDL), as long as interfacing signals and their timing are correctly defined. Concerning the task of security property verification, however, treating a sub-module as a functional unit by ignoring its internal structure may cause problems. Security properties that are proven for the top level module and all its sub-modules do not

guarantee that the same properties will hold for the whole hierarchical design, where attackers can easily insert hardware Trojans to maliciously modify the interface without violating security properties proven for all modules separately. As a result, the operation of module definition/instantiation should be defined in a way that the details of sub-modules are accessible from the top level module so that any security properties, if proven, remain valid for the whole design. Thus, in PCH we flatten the hierarchical design such that the sub-modules and their interfaces are transparent to the top module. *module* and *module-inst* are key words for module definitions and instantiations. In [53], a new syntax for representing modules is introduced in Coq, which preserves the hierarchical structure and does not require design flattening.

The underlying formal language of the Coq proof assistant, *Gallina*, is based on dependently typed lambda calculus and it defines both types and terms in the same syntactical structure. During the translation process, syntax and semantics of the HDL are translated to *Gallina* using the *formal-HDL*.

10.3.2 Data Protection Through Information Flow Tracking

Among all potential RT-level malicious modifications, sensitive information protection has been a research topic within the cybersecurity domain for decades. Various approaches have been developed, relying on safe languages and software level dynamic checks, to detect buffer overflow attacks and format string vulnerabilities. These methods suffer from the limitation that they either have high false-alarm rates or would cause significant performance overhead. Taking these limitations into consideration, researchers invented new information protection schemes based on hardware–software co-design, where the hardware infrastructure is actively involved in dynamic information flow tracking. This new trend has proven successful in improving detection accuracy and lowering performance overhead, at the cost of hardware level modifications. For example, authors in [54] proposed a dynamic information flow tracking framework with all internal storage elements equipped with a security tag.

Authors in [55] focused on pointer tainting to prevent both control data and non-control data attacks. Besides information flow tracking, the hardware is also enhanced to help prohibit information leakage, such as in the InfoShield architecture [56], which applies restrictions to operations on sensitive data. Similarly, the RIFLE architecture is developed on top of an information flow security (IFS) instruction set architecture (ISA), where all states defined by the base ISA are augmented by labels [57]. More recently, a new software–hardware architecture was developed to support more flexible security policies, either to protect sensitive data [58] or to prevent malicious operations from untrusted third-party OS kernel extensions [59].

Two formal information flow tracking methodologies were also developed, namely *static information flow tracking* [60] and *dynamic information assurance*

[9], which address the challenge of hardware Trojans, capable of leaking sensitive information. These two schemes follow the concept of proof-carrying hardware IP (PCHIP) [8] to enhance the trustworthiness of third-party IP cores.

These two formal methods are particularly geared toward secret information protection, counteracting RTL hardware Trojan attacks in hardware soft IPs, and preventing unintended design backdoors. These two methods differ in complexity and the approach they take to track information in the design. Static information flow tracking scheme is suitable for small designs, and requires less effort in proof development, while dynamic information assurance scheme considers the requirements of more complex and pipelined designs, and needs much more effort in constructing the proofs of security theorems. Designers can adopt these two methodologies based on their requirements.

The *static information flow tracking scheme* and the *dynamic information assurance scheme* are integrated with the PCH IP protection framework and they accept the data secrecy properties as the security property. Furthermore, because the target data secrecy properties are independent of circuit functional specifications, IP vendors may translate the properties from natural language to formal theorems without specifying target circuits and can store the translated formal theorems in a property library for similar designs. The development of a Coq property library and the reuse of theorem-proof contents lowers the burden for IP vendors and stimulates wider acceptance of the proposed proof-carrying based hardware IP protection method. Property formalization and proof generation of both schemes are performed using the Coq proof assistant platform [11].

10.3.2.1 Static Information Flow Tracking Scheme

For the static information flow tracking scheme [60], the IP vendor first designs the circuit based on the functional specifications provided by the IP consumer, in the form of HDL codes. Utilizing a formal semantic model and static information flow tracking rules, the IP vendor then converts the circuit from HDL code into formal logic. In parallel, the IP vendor uses the property formalization constraints to translate the agreed-upon data secrecy properties from natural language to formal theorems. The IP vendor will then try to construct proofs for the translated theorems within the context of the target circuit. Even though the IP vendor is responsible for both circuit design and theorem proving, given a set of well-defined theorems, it is not possible to prove the theorems with a Trojan-infected circuit containing the prohibited information leakage paths. Both formal theorems and their proofs are part of the final deliverable handed to the IP consumer.

Upon receiving the hardware bundle which includes the HDL code and theorem-proof pairs for data secrecy properties, the IP consumer regenerates the formal logic of the original circuit based on the same formal semantic model and static information flow tracking rules. The IP consumer also checks whether the security theorems (in formal language) accurately represent the data secrecy properties (in

natural language). The security theorems and related proofs will then be combined with the regenerated formal logic to pass through an automatic proof checker. If no exceptions are raised, then we claim that the delivered IP core fulfills the agreed-upon data secrecy properties. However, any errors during the proof checking process warn the user that malicious circuits (or design flaws) may exist in the IP core, making it violate the data secrecy properties.

10.3.2.2 Dynamic Information Assurance Scheme

The static scheme is effective in detecting data leakage caused by hardware Trojans and/or design faults. It also requires less effort for constructing proofs. However, the static scheme is limited by the fact that it can only check circuit trustworthiness statically. To overcome this shortcoming of the static scheme and to achieve high-level hardware Trojan detection capability, a dynamic information assurance scheme is later developed [9].

This dynamic scheme supports various levels of circuit architectures, ranging from low-complexity single-stage designs to large-scale deeply pipelined circuits. Similar to the static scheme, the dynamic scheme also focuses on circuits dealing with sensitive information, such as cryptographic designs, because it sets data secrecy as the primary goal and tries to prevent illegal information leakage from IP cores. Within the dynamic scheme, all signals are assigned values indicating their sensitivity levels. These values will be updated after each clock cycle according to their original values and the updating rules defined by the signal sensitivity transition model. Since the sensitivities of all circuit signals are managed in a sensitivity list, two sensitivity lists are of interests for data secrecy protection: the initial sensitivity list and the stable sensitivity list. The initial sensitivity list reflects the circuit status after initialization or powered-on mode when only some input signals contain sensitive information, such as plaintext and encryption keys. The stable sensitivity list, on the other hand, indicates the circuit status when all internal/output signals are of fixed sensitivity levels.

Similar to the static scheme, IP vendor will also translate the agreed-upon data secrecy properties from natural language to property generation functions, which can later help to generate formal theorems. Meanwhile, different from the static scheme, IP consumers will first check the contents of the initial signal sensitivity list and the stable signal sensitivity list, which represent the circuit's initial secrecy status and the stabilized status, respectively. The validity of the initial list is checked to ensure that sensitivity levels are appropriately assigned to all input/output/internal signals. The circuit's stable sensitivity status contains complete information of the distribution of sensitive information across the whole circuit, so the stable list will then be carefully evaluated to detect any backdoors that may leak sensitive information. After both signal sensitivity lists pass the initial checking, IP consumers proceed to the next step of proof checking. A "PASS" output from the automatic proof checker provides evidence that HDL codes do not contain any

malicious channels to leak information. However, a "FAIL" result is a warning that some of the data secrecy properties are breached in the delivered IP cores.

10.3.3 Hierarchy Preserving Verification

The above mentioned PCH frameworks treat the whole circuit design as one module and prove security properties on them [8–10, 60]. That is, the entire design is first flattened before translating the HDL code of the design into the formal language and proving it with respect to formal security theorems. Design flattening increases the complexity of translating HDL code into *Gallina*. It also adds to the risk of introducing errors during the code conversion process. Due to flattening, a verification expert has to go through the entire design in order to construct proofs of security theorems, which significantly increases the workload for design verification. Also, any updates to the HDL code will significantly change the proof for the same security property. Moreover, the PCH framework prevents proof reuse, i.e., proofs constructed for one design cannot be used in another design even though the same IP modules are used. All of these limitations prohibit a wide usage of the PCH framework in modern SoC designs.

To overcome these limitations, the *Hierarchy-preserving Formal Verification* (HiFV) framework is developed for verifying security properties on SoC designs in [53]. The HiFV framework is an extension of the PCH framework. In the HiFV framework, the design hierarchy of the SoC is preserved and a distributed approach is developed for constructing proofs of security properties. In the distributed approach, security properties are divided into sub-properties in such a way that each sub-property corresponds to an IP module of the SoC. Proofs are then constructed for these sub-properties and the security property for the SoC design is proven through the integration of all proofs from sub-properties. Similar to PCH, the HiFV framework requires semantic translation of the HDL code and informal security properties to *Gallina*. For proving the trustworthiness of the HDL code of the SoC, Hoare-logic is used. Similar to other PCH methods, the HiFV framework is carried out in Coq.

As mentioned earlier, before building the formal model for the SoC system, the syntax and semantics should be defined and then shared by any parties who need to design or check the proof. In addition, *interface* and *module* are incorporated in the *formal-HDL* to preserve the design hierarchy of the SoC. That is, in order to make distributed proof construction applicable on hierarchical designs, an *interface* is developed in the HiFV framework which makes the verification process flexible and efficient for the proof writer. To define the *interface*, information about each IP and its corresponding I/O are needed, such as the name, number, and data type. By using the *interface*, the management of the plenty of formal modules would be much easier in the verification house side. The structure of the *interface* is shown in Fig. 10.4. Through the *interface*, an IP module within an SoC can access other

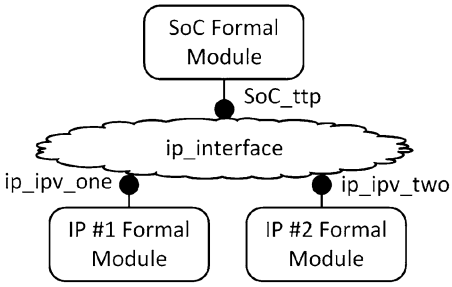


Fig. 10.4 Structure of the SoC with *interface*

modules such as IP #1 Formal Module or IP #2 Formal Module in the figure. The *ip_ipv_one*, *ip_ipv_two*, and *SoC_ttp* are the name of the corresponding *interfaces*.

The distributed proof construction process uses Hoare-logic, where the trustworthiness of the SoC *formal-HDL* code is determined by ensuring that the code operates following the constraints of the pre-condition and the post-condition. The pre-condition of the *formal-HDL* code is the initial configuration of the design and the post-condition is the security theorem. Meanwhile, in order to overcome the scalability issue, a distributed proof construction approach is developed, which is dedicated for SoC designs with hierarchical structures. This approach makes the HiFV framework scalable by reducing the time required for proof construction, proof correction, and proof modification.

In the HiFV framework, the translated HDL code of the SoC, formal security theorems, and the initial configuration of the design is represented as a Hoare Triple (Eq. (10.1)).

$$(\phi)CoqEquivalentCode\_SoC(\psi) \tag{10.1}$$

In this equation, ϕ is the pre-condition corresponding to the initial configuration of the design. The translated HDL code of the SoC design hierarchy in *Gallina* is given by *CoqEquivalentCode_SoC*. In the process of translation, modules in the SoC HDL code, which correspond to IPs from different vendors, are also translated. The post-condition is given by ψ which represents the formal security theorem.

The security theorem is divided into lemmas (Eq. (10.2)), which are post-conditions for individual IP modules. In Eq. (10.2), post-condition for IPs (lemmas) are represented as ψ_i $(1 \leq i \leq n)$, $n = maximum\ number\ of\ IP\ modules\ required\ to\ prove\ the\ security\ theorem$ and ψ is the security theorem. These lemmas correspond to those IP modules that are required to satisfy the security theorem.

$$\psi := \psi_1 \wedge \psi_2 \cdots \wedge \psi_n \tag{10.2}$$

Similarly, the pre-condition of the SoC design (ϕ) and the translated HDL code of the SoC design (*CoqEquivalentCode_SoC*) are divided according to Eqs. (10.3) and (10.4). Here, (ϕ_i) and (*CoqEquivalentCode_IPmodule_i*) $(1 \leq i \leq n)$ represent the pre-conditions and translated HDL code of each IP module of the

SoC, respectively.

$$\phi := \phi_1 \wedge \phi_2 \cdots \wedge \phi_n \tag{10.3}$$

$$\begin{aligned} CoqEquivalentCode\_SoC := {} & CoqEquivalentCode\_IPmodule\_1 \\ & \wedge CoqEquivalentCode\_IPmodule\_2 \ldots \\ & \wedge CoqEquivalentCode\_IPmodule\_n \end{aligned} \tag{10.4}$$

The HDL code of the IP core is certified to be trustworthy only if it satisfies the pre-condition and the post-condition. When all the modules of IP cores satisfy the post-conditions (lemmas), we can state that the security theorem is proven for the SoC design.

$$(\phi_i) CoqEquivalentCode\_IPmodule\_i(\psi_i) \tag{10.5}$$

The distributed approach of proof construction also enables proof reuse. After certifying the trustworthiness of each IP core of the SoC, the proofs can be stored in a library and accessed by the trusted third party (TTP) verification house for verification of other SoC designs in which the same IP modules are used and similar security properties are applied. In this way the HiFV framework further reduces the time for verifying complex designs.

As a summary, in this approach, the previously developed PCH framework is extended into the SoC design flow and largely simplified the process for proving security properties through a hierarchical proof construction procedure. To reduce the workload for circuit verification, the proof of the security properties for individual IPs can be encapsulated and reused in proving security properties at the SoC level. Also, in the hierarchical framework, the amount of updates that need to be done to existing proofs when SoC designs are modified is significantly lowered. The developed HiFV framework paves the way for large-scale circuit design security verification.

10.3.4 Integrating Theorem Prover and Model Checker

Although the HiFV hierarchical approach improves scalability of the previous PCH method, it still suffers from the challenge of proof construction. Meanwhile, model checkers such as Cadence IFV cannot be used for verifying systems with large state space because of the space explosion problem. As the number of state variables (n) in the system increases, amount of space required for representing the system and the time required for checking the system increases exponentially ($T(n) = 2^{O(n)}$) (Fig. 10.5).

To further overcome the scalability issue and to verify a computer system, an *integrated formal verification framework* (see Fig. 10.6) is introduced in [61], where

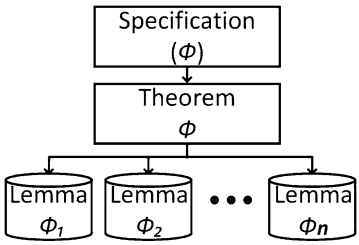


Fig. 10.5 Security specification (ϕ) decomposed into lemmas

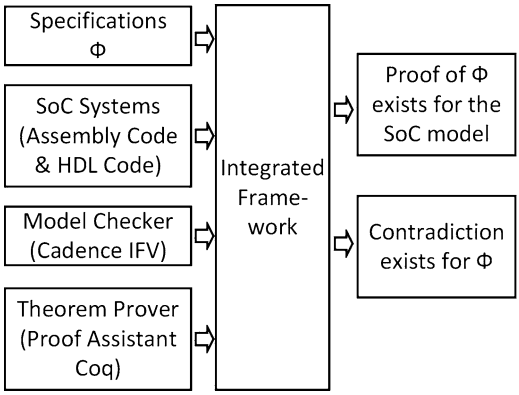


Fig. 10.6 Integrated formal verification framework

the security properties are checked against SoC designs. In this framework, the theorem prover is combined with a model checker for proving formal security properties (specifications). Moreover, the hierarchical structure of the SoC is leveraged to reduce the verification effort.

Some efforts have been made to combine theorem provers with model checkers for verification of hardware and software systems [62, 63]. These methods try to overcome the scalability issue of both techniques. That is, both model checkers and theorem provers cannot scale well to formally verify large-scale circuit designs. Some of the popular theorem provers such as higher order logic (HOL Light) and prototype verification system have integrated model checkers. These tools have been used for functional verification of hardware systems. For the first time, this combined technique has been extended toward verification of security properties on third-party IP cores and SoCs [61].

In the integrated framework, the hardware design, represented in a hardware description language (HDL), and the assembly level instructions of a vulnerable program, is first translated to *Gallina*, which is similar to other PCH methods. Then, the security specification is stated as a formal theorem in Coq. In the following step, this theorem is decomposed into disjoint lemmas (see Fig. 10.5) based on sub-modules. These lemmas are then represented in the Property Specification Language (PSL) specification language and are called sub-specifications. Subsequently, the Cadence IFV verifies the sub-modules against the corresponding sub-specifications. Sub-modules are functions, which have less number of state variables and are

connected to primary output of the design. These functions are always from the bottom level of SoC and have rare dependency relationship with each other.

The HDL code of a large design consists of many such sub-modules. If the sub-modules satisfy the sub-specifications, lemmas are considered to be proved. Checking the truth value of the sub-specifications with a model checker eliminates the effort required for proving the lemmas and translating the sub-modules to Coq. Upon proving these sub-modules, Hoare-logic is then used to combine proof of these lemmas to prove the security theorem of the entire system in Coq.

The integrated formal verification framework helps in protecting a large-scale SoC design from malicious attacks. Given that an interactive theorem prover (e.g., Coq) requires lots of effort to manually verify the design and that a model checker suffers from scalability issues, these two techniques are combined together through the decomposition of the security property as well as the design in such a way that the model checker can verify those sub-modules which have much less state variables. Consequently, the amount of effort required for translating the design from HDL to *Gallina* and proving the security theorem in Coq is reduced.

10.4 Conclusion

In this chapter, we explain our interactive theorem proving based PCH approach for security property verification of hardware IP cores. We also describe application of the framework for preventing information leakage from soft IPs. To overcome scalability and reusability issues of original PCH method, a design hierarchy preserving scheme was then introduced that incorporates both model checking and interactive theorem proving for verification.

Acknowledgements This work has been partially supported by the National Science Foundation (NSF-1319105), the Army Research Office (ARO W911NF-16-1-0124), and Cisco.

References

1. M. Banga, M. Hsiao, Trusted RTL: Trojan detection methodology in pre-silicon designs, in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2010), pp. 56–59
2. A. Waksman, M. Suozzo, S. Sethumadhavan, FANCI: identification of stealthy malicious logic using boolean functional analysis, in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, CCS'13* (2013), pp. 697–708
3. D. Sullivan, J. Biggers, G. Zhu, S. Zhang, Y. Jin, FIGHT-metric: Functional identification of gate-level hardware trustworthiness, in *Design Automation Conference (DAC)* (2014)
4. N. Tsoutsos, C. Konstantinou, M. Maniatakos, Advanced techniques for designing stealthy hardware trojans, in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (2014)

5. M. Rudra, N. Daniel, V. Nagoorkar, D. Hoe, Designing stealthy trojans with sequential logic: A stream cipher case study, in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (2014)
6. S. Drzevitzky, U. Kastens, M. Platzner, Proof-carrying hardware: Towards runtime verification of reconfigurable modules, in *International Conference on Reconfigurable Computing and FPGAs* (2009), pp. 189–194
7. S. Drzevitzky, M. Platzner, Achieving hardware security for reconfigurable systems on chip by a proof-carrying code approach, in *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip* (2011), pp. 1–8
8. E. Love, Y. Jin, Y. Makris, Proof-carrying hardware intellectual property: a pathway to trusted module acquisition. IEEE Trans. Inf. Forensics Secur. **7**(1), 25–40 (2012)
9. Y. Jin, B. Yang, Y. Makris, Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing, in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2013), pp. 99–106
10. Y. Jin, Y. Makris, A proof-carrying based framework for trusted microprocessor IP, in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2013), pp. 824–829
11. INRIA, The Coq proof assistant (2010), http://coq.inria.fr/
12. F. Wolff, C. Papachristou, S. Bhunia, R.S. Chakraborty, Towards Trojan-free trusted ICs: problem analysis and detection scheme, in *IEEE Design Automation and Test in Europe* (2008), pp. 1362–1365
13. M. Hicks, M. Finnicum, S.T. King, M.M.K. Martin, J.M. Smith, Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *Proceedings of IEEE Symposium on Security and Privacy* (2010), pp. 159–172
14. C. Sturton, M. Hicks, D. Wagner, S. King, Defeating UCI: building stealthy and malicious hardware, in *2011 IEEE Symposium on Security and Privacy (SP)* (2011), pp. 64–77
15. X. Zhang, M. Tehranipoor, Case study: detecting hardware trojans in third-party digital ip cores, in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2011), pp. 67–70
16. Y. Jin, Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits, in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2014), pp. 19–24
17. X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, P. Mishra, Pre-silicon security verification and validation: a formal perspective, in *Proceedings of the 52Nd Annual Design Automation Conference, DAC'15* (2015), pp. 145:1–145:6
18. F.M. De Paula, M. Gort, A.J. Hu, S.J. Wilton, J. Yang, Backspace: formal analysis for post-silicon debug, in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design* (IEEE Press, New York, 2008), p. 5
19. S. Drzevitzky, Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration, in *2010 International Conference on Field Programmable Logic and Applications (FPL)* (2010), pp. 255–258
20. J. Rajendran, V. Vedula, R. Karri, Detecting malicious modifications of data in third-party intellectual property cores, in *Proceedings of the Annual Design Automation Conference, DAC '15* (ACM, New York, 2015), pp. 112:1–112:6
21. J. Harrison, Floating-point verification, in *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings*, ed. by J. Fitzgerald, I.J. Hayes, A. Tarlecki. Lecture Notes in Computer Science, vol. 3582 (Springer, Berlin, 2005), pp. 529–532
22. S. Owre, J.M. Rushby, N. Shankar, PVS: a prototype verification system, in *11th International Conference on Automated Deduction (CADE)* (Saratoga, NY), ed. by D. Kapur. Lecture Notes in Artificial Intelligence, vol. 607 (Springer, Berlin, 1992), pp. 748–752
23. D. Russinoff, M. Kaufmann, E. Smith, R. Sumners, Formal verification of floating-point RTL at AMD using the ACL2 theorem prover, in *Proceedings of the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation, Paris, France* (2005)
24. J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, A. Platzer, How to model and prove hybrid systems with KeYmaera: a tutorial on safety. Int. J. Softw. Tools Technol. Transfer **18**, 67–91 (2016)

25. A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant* (MIT Press, Cambridge, 2013)
26. U. Norell, Dependently typed programming in Agda, in *Advanced Functional Programming* (Springer, Berlin, 2009), pp. 230–266
27. R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, S.F. Smith, *Implementing Mathematics with the Nuprl Proof Development System* (Prentice-Hall, Upper Saddle River, 1986)
28. L.C. Paulson, Isabelle: the next 700 theorem provers, in *Logic and Computer Science*, vol. 31 (Academic Press, London, 1990), pp. 361–386
29. E.M. Clarke, O. Grumberg, D. Peled, *Model Checking* (MIT press, Cambridge, 1999)
30. T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Software verification with blast, in *Model Checking Software*, (Springer, Berlin, 2003), pp. 235–239
31. J. O'Leary, X. Zhao, R. Gerth, C.-J.H. Seger, Formally verifying ieee compliance of floating-point hardware. Intel Technol. J. **3**(1), 1–14 (1999)
32. M. Srivas, M. Bickford, Formal verification of a pipelined microprocessor. IEEE Softw. **7**(5), 52–64 (1990)
33. T. Kropf, Introduction to Formal Hardware Verification (Springer, Berlin, 2013)
34. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, seL4: formal verification of an os kernel, in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles* (ACM, New York, 2009), pp. 207–220
35. S. Chaki, E.M. Clarke, A. Groce, S. Jha, H. Veith, Modular verification of software components in C. IEEE Trans. Softw. Eng. **30**(6), 388–402 (2004)
36. H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M.F. Kaashoek, N. Zeldovich, Using crash hoare logic for certifying the fscq file system, in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP'15* (ACM, New York, 2015), pp. 18–37
37. M. Vijayaraghavan, A. Chlipala, N. Dave, Modular deductive verification of multiprocessor hardware designs, in *Computer Aided Verification* (Springer, Cham, 2015), pp. 109–127
38. A.A. Mir, S. Balakrishnan, S. Tahar, Modeling and verification of embedded systems using cadence SMV, in *2000 Canadian Conference on Electrical and Computer Engineering*, vol. 1 (IEEE, New York, 2000), pp. 179–183
39. M. Kwiatkowska, G. Norman, D. Parker, Prism: probabilistic symbolic model checker, in *Computer Performance Evaluation: Modelling Techniques and Tools* (Springer, Berlin, 2002), pp. 200–204
40. G.J. Holzmann, The model checker spin. IEEE Trans. Softw. Eng. **23**(5), 279 (1997)
41. D. Beyer, M.E. Keremoglu, Cpachecker: a tool for configurable software verification, in *Computer Aided Verification* (Springer, Berlin, 2011), pp. 184–190
42. A. David, K. G. Larsen, A. Legay, M. Mikučionis, Z. Wang, Time for statistical model checking of real-time systems, in *Computer Aided Verification* (Springer, Berlin, 2011), pp. 349–355
43. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in *Computer Aided Verification*, (Springer, Berlin 2000), pp. 154–169
44. C. Baier, J. Katoen, *Principles of Model Checking* (MIT Press, Cambridge, 2008)
45. A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using sat procedures instead of BDDs, in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference* (ACM, New York, 1999), pp. 317–320
46. R.E. Bryant, Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
47. R.E. Bryant, Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **100**(8), 677–691 (1986)
48. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: an opensource tool for symbolic model checking, in *Computer Aided Verification* (Springer, Berlin, 2002), pp. 359–364

49. E. Clarke, A. Biere, R. Raimi, Y. Zhu, Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
50. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, Y. Zhu, Bounded model checking Adv. Comput. **58**, 117–148 (2003)
51. S. Qadeer, J. Rehof, Context-bounded model checking of concurrent software, in *Tools and Algorithms for the Construction and Analysis of Systems* (Springer, Berlin, 2005), pp. 93–107
52. G.C. Necula, Proof-carrying code, in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), pp. 106–119
53. X. Guo, R.G. Dutta, Y. Jin, Hierarchy-preserving formal verification methods for pre-silicon security assurance, in *16th International Workshop on Microprocessor and SOC Test and Verification (MTV)* (2015)
54. G.E. Suh, J.W. Lee, D. Zhang, S. Devadas, Secure program execution via dynamic information flow tracking, in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI (2004), pp. 85–96
55. S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, R. Iyer, Defeating memory corruption attacks via pointer taintedness detection, in *Proceedings. International Conference on Dependable Systems and Networks, 2005. DSN 2005* (2005), pp. 378–387
56. W. Shi, J. Fryman, G. Gu, H.-H. Lee, Y. Zhang, J. Yang, Infoshield: a security architecture for protecting information usage in memory, in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006* (2006), pp. 222–231
57. N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, D. August, RIFLE: an architectural framework for user-centric information-flow security, in *37th International Symposium on Microarchitecture, 2004. MICRO-37 2004* (2004), pp. 243–254
58. Y.-Y. Chen, P. A. Jamkhedkar, R.B. Lee, A software-hardware architecture for self-protecting data, in *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS'12* (2012), pp. 14–27
59. Y. Jin, D. Oliveira, Extended abstract: trustworthy SoC architecture with on-demand security policies and HW-SW cooperation, in *5th Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-5)* (2014)
60. Y. Jin, Y. Makris, Proof carrying-based information flow tracking for data secrecy protection and hardware trust, in *IEEE 30th VLSI Test Symposium (VTS)* (2012), pp. 252–257
61. X. Guo, R.G. Dutta, P. Mishra, Y. Jin, Scalable soc trust verification using integrated theorem proving and model checking, in *IEEE Symposium on Hardware Oriented Security and Trust (HOST)* (2016), pp. 124–129.
62. S. Berezin, *Model checking and theorem proving: a unified framework*. Ph.D. Thesis, SRI International (2002)
63. P. Dybjer, Q. Haiyan, M. Takeyama, Verifying haskell programs by combining testing, model checking and interactive theorem proving. Inf. Softw. Technol. **46**(15), 1011–1025 (2004)

Chapter 11
Hardware Trust Verification

Qiang Xu and Lingxiao Wei

11.1 Introduction

Hardware Trojans (HTs) are malicious alterations of normal integrated circuits (IC) designs. They have been a serious concern due to the ever-increasing hardware complexity and the large number of third-parties involved in the design and fabrication process of ICs.

For example, a hardware backdoor can be introduced into the design by simply writing a few lines of hardware description language (HDL) codes [1, 2], which leads to functional deviation from design specification and/or sensitive information leakages. Skorobogatov and Woods [3] found a "backdoor" in a military-grade FPGA device, which could be exploited by attackers to extract all the configuration data from the chip and access/modify sensitive information. Liu et al. [4] demonstrated a silicon implementation of a wireless cryptographic chip with an embedded HT and showed it could leak secret keys. HTs thus pose a serious threat to the security of computing systems and have called upon the attention of several government agencies [5, 6].

HTs can be inserted in ICs in almost any stage, e.g., specification, register-transfer level (RTL) design, logic synthesis, intellectual property (IP) core integration, physical design, and manufacturing process. Generally speaking, the likelihood of HTs being inserted at design time is usually much higher than that being inserted at manufacturing stage, because adversaries do not need to access foundry facilities to implement HTs and it is also more flexible for them to implement various malicious functions. Hardware designs can be covertly compromised by HTs inserted into the RTL code or netlist. These HTs may be implemented by rogue

Q. Xu (✉) • L. Wei
CUhk REliable computing laboratory (CURE Lab.), Department of Computer Science & Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong
e-mail: qxu@cse.cuhk.edu.hk; lxwei@cse.cuhk.edu.hk

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_11

designers in the development team or integrated from a malicious third-party IP core. As it is not feasible to verify trustworthiness of ICs at fabrication stage, it is important to ensure that the design is HT-free before it enters the next phase of IC design cycle.

11.2 HT Classification

Generally speaking, a HT is composed of its activation mechanism (referred to as *trigger*) and its malicious function (referred to as *payload*). In order to pass functional test and verification, stealthy HTs usually employ certain trigger condition that is controlled by dedicated trigger inputs and difficult to be activated with verification test cases.

The HTs inserted at design stage can be categorized according to their impact on the normal functionalities of original circuits. Trojans can either directly modify the normal functions or insert extra logic to introduce additional malicious behavior while maintaining original functionality.

They are named *bug-based HT* and *parasite-based HT*, respectively.

11.2.1 Bug-Based HT

A bug-based HT changes the circuit in a manner that causes it to lose some of its normal functionalities. Consider an original design in Fig. 11.1a whose normal function is $f_n = d_1 d_2$. An attacker may change it to a malicious function, $f_m = \bar{d}_1 d_2$, by adding an additional inverter, as shown in Fig. 11.1b. With this malicious change, the circuit has lost certain functionalities, i.e., the two circuits behave differently when $d_2 = 1$. Their corresponding K-Maps are shown in Fig. 11.2a, b. By comparing the two K-Maps, we can observe that some entries of the normal function have been modified by the malicious function as highlighted in gray.

For bug-based HT, some functional inputs serve as trigger inputs to the HT, e.g., for the circuit shown in Fig. 11.1b, d_2 is both a functional input and a trigger input.

From a different perspective, the bug-based HT can be simply regarded as a design bug (with malicious intention though), as the design in fact does not realize all of its normal functionalities designated by the specification. As a result, the extensive simulation/emulation is likely to detect this type of HTs. From this perspective, bug-based HT is usually not a good choice for attackers in terms of the stealthy requirement, and almost all HT designs appeared in the literature (e.g., [1, 2, 7–11]) belong to the parasite-based type, as discussed in the following.

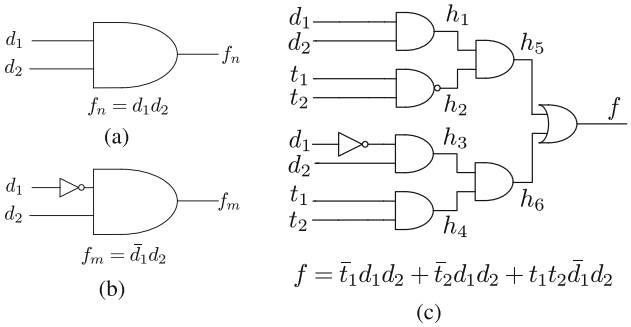


Fig. 11.1 HT classification with a simple example. (**a**) Original circuit. (**b**) Bug-based HT. (**c**) Parasite-based HT

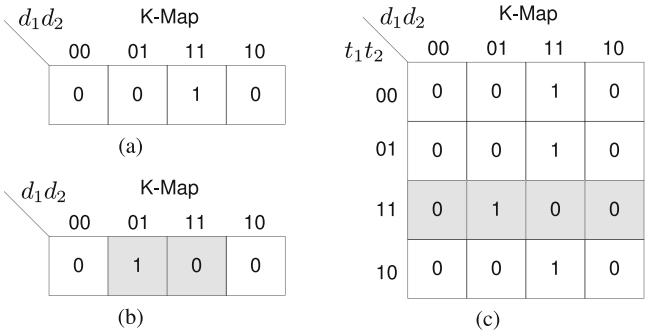


Fig. 11.2 (**a**) K-Map of original circuit in Fig. 11.1a; (**b**) K-Map of bug-based HT in Fig. 11.1b; (**c**) K-Map of parasite-based HT in Fig. 11.1c

11.2.2 Parasite-Based HT

A parasite-based HT exists along with the original circuit, and does not cause the original design to lose *any* normal functionalities. Again, consider an original circuit whose normal function is $f_n = d_1 d_2$. Suppose an attacker wants to insert a HT whose malicious function is $f_m = \bar{d}_1 d_2$ into the design. To control when the design runs the normal function and when it runs malicious function, the attacker could employ some additional inputs as trigger inputs, t_1 and t_2. Usually in order to escape the functional verification, trigger inputs are carefully selected and the trigger condition is designed to be an extremely rare event occurred with verification tests.

Let us examine the K-Map of the parasite-based HT-inserted circuit, as shown in Fig. 11.2c. The third row represents the malicious function while other rows show the normal function. By comparing it with the K-Map of the original circuit (see Fig. 11.2a), we can observe that the parasite-based HT enlarges the K-Map size with additional inputs so that it can keep the original function while embedding the malicious function. The circuit can then perform the normal function and the malicious function alternately, controlled by trigger inputs.

11.3 Verification Techniques for Hardware Trust

11.3.1 Functional Verification

Ideally, a HT can be detected by activating it and observing its malicious behaviors. During functional verification, a set of test cases would be applied to verify the functional correctness of the targeted design. As bug-based HTs utilize some functional inputs as triggers, it is highly likely to be activated and detected by extensive simulation in a functional verification process. Hence it is not good choice for adversaries. For parasite-based HTs, in theory, it can also be activated by enumerating all possible system state in an exhaust simulation. In reality, however, even with the sheer volume of states that exist in a simple design, functional verification can only cover a small subset of the functional space of a hardware design. Considering the fact that attackers have full controllability for the location and the trigger condition of their HT designs at design time, which are secrets to functional verification engineers, it is usually very difficult, if not impossible, to directly activate a parasite-based HT.

11.3.2 Formal Verification

Theoretically speaking, whether a hardware design contains HTs or not can be theoretically proven when a given trustworthy high-level system model (e.g., [12]) with formal verification. In practice, however, full formal verification of large circuits is still computationally infeasible. In addition, the golden model itself may not be available. Consequently, it is not appropriate to adopt formal verification to verify hardware trust and dedicated trust verification techniques are developed for parasite-based HT detection.

11.3.3 Trust Verification

Trust verification techniques flag suspicious circuitries based on the observation that HTs are nearly always dormant (by design) in order to pass functional verification. Such analysis can be conducted in a *dynamic* manner by analyzing which part of the circuit is not sensitized during functional verification, as in *UCI* [10] and VeriTrust [13]. Alternatively, *static* Boolean functional analysis can be used to identify suspicious signals with *weakly affecting* inputs, as in FANCI [14].

11.3.3.1 Unused Circuit Identification

Hicks et al. [10] first addressed the problem of identifying HTs inserted at design time, by formulating it as an unused circuit identification problem. The primary assumption in UCI is the hardware designs usually are verified with an extensive test suite while the HTs inside can successfully avoid being activated to manifest certain malicious behaviors on the outputs. Hence the "unused circuits," which are included in the design but do not affect any output during verification, are considered to be HT candidates.

One way to define unused circuit is based on the code coverage metrics used in verification (e.g., line coverage and branch coverage) such that those uncovered circuitries are flagged as suspicious malicious logic. However, it is not hard for attackers to craft circuits that are covered by verification but the trojan is never triggered. Suppose a HT is inserted into a HDL implementation as follows:

```
X <= (ctrl[0] == '0')?normal_func:trojan_func;
Y <= (ctrl[1] == '0')?normal_func:trojan_func;
Out <= (ctrl[0] == '0')?X:Y;
```

If the control values 00, 01, and 10 are included in the verification process, all three lines in the above code will be covered, but the out signal will always equal to the normal function. This simple definition based on coverage is easily defeated thus not appropriate for HT detection.

They defined "unused circuits" as follows. Consider a signal pair (s, t), where t is dependent on s. If $t = s$ throughout the entire functional verification procedure, the intermediate circuit between s and t is regarded as "unused circuit."

The UCI algorithm is performed in two steps. First, a data-flow graph is generated in which nodes are signals or state elements and edges indicate data flow between nodes. Based on the data-flow graph, a list of signal pairs or (data-flow pairs), where data flows from a source signal to a sink signal, are generated. The list includes both direct dependencies and indirect dependencies. Second, the HDL code is simulated with verification tests to detect the signal pairs that do not have data flows. On each simulation step, UCI checks inequality for each remaining data-flow pairs. If inequality is detected, the signal pair is regarded safe and are removed from suspicious list. Finally, after the simulation completes, a set of remaining data-flow pairs is identified in which the logic between the nodes does not affect the signal value from source to sink.

For the example circuit shown in Fig. 11.3a, a HT is inserted with malicious function $f_m = t_1 t_2 d_2$ with trigger condition $\{t_1, t_2\} = \{1, 1\}$. The data-flow graph of this example circuit is shown in Fig. 11.3b. From the data-flow graph, a list of 11 signal pairs can be constructed in the left part of Table 11.1.

Simulation is performed by applying test cases except those containing $\{t_1, t_2\} = \{1, 1\}$, which would activate the HT. Every signal pair's equality is recorded during simulation and their results are shown in Table 11.1. The signal pair (h_2, f) is always

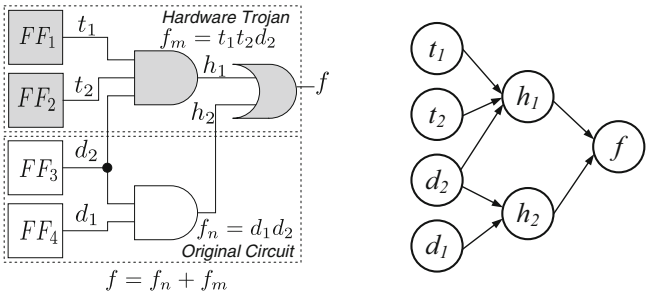


Fig. 11.3 (**a**) A HT-infected circuit with trigger inputs t_1 and t_2; (**b**) Data-flow graph of the HT-infected circuit

Table 11.1 Signal pairs generated from UCI

Signal pairs		Always equal under
Source	Sink	non-trigger condition
t_1	h_1	No
t_2	h_1	No
d_1	h_1	No
d_1	h_2	No
d_2	h_2	No
h_1	f	No
h_2	f	Yes
t_1	f	No
t_2	f	No
t_1	f	No
t_2	f	No

equal under non-trigger condition as h_1 is always 0. Hence the OR gate between them is regarded as "unused circuit" and is suspicious of HT infected.

With the above shown algorithm, given a specific hardware design in HDL code, the UCI algorithm in [10] traces all signal pairs during verification, and reports those ones for which the property $s = t$ holds throughout all test cases as suspicious circuitries.

One of the main limitations of UCI techniques is that they are sensitive to the implementation style of HTs. Later, Sect. 11.4.1.1 presented how to exploit this weakness to defeat UCI detection algorithms.

11.3.3.2 VeriTrust

In order to conquer the limitations of UCI, Zhang et al. proposed a HT detection method called *VeriTrust*. VeriTrust [13] flags suspicious circuitries by identifying potential trigger inputs used in HTs, based on the observation that these inputs

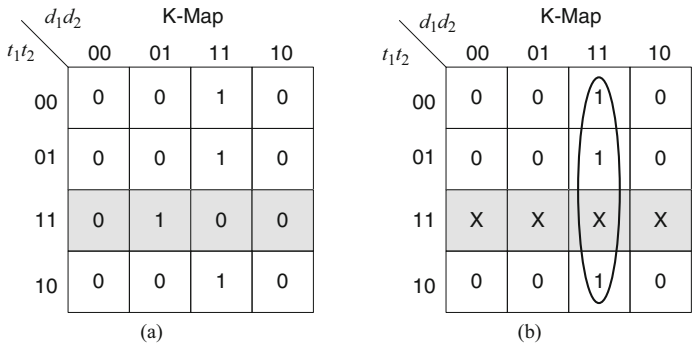


Fig. 11.4 (**a**) A HT-infected circuit with trigger inputs t_1 and t_2; (**b**) Data-flow graph of the HT-infected circuit

keep dormant under non-trigger condition (otherwise HTs would have manifested themselves) and hence are **redundant** to the normal logic function of the circuit.

The intuition behind *VeriTrust* is the fact that any HT-infected signal must be driven by at least one dedicated trigger input. Thus the function of a HT-infected signal can be represented as $f = C_n f_n + C_m f_m$, wherein f_n and f_m denote the circuit's normal function and HT's malicious function while C_n is non-triggering condition and C_m the triggering condition, respectively. As C_m never appear in the verification process, all its entries in K-map can be set to don't-cares without affecting the normal function. After logic simplification, the function $f = f_n$, which means the triggering inputs, becomes redundant.

Here is an illustration based on the example circuit from Fig. 11.1c. Its Boolean function is $f = \bar{t}_1 d_1 d_2 + \bar{t}_2 d_1 d_2 + t_1 t_2 \bar{d}_1 d_2$. The K-map of this function is shown in Fig. 11.4a. Suppose the verification has verified every entries in K-map except those with triggering condition, highlighted with gray color in Fig. 11.4a. These entries would be set to don't-cares in *VeriTrust*, as shown in Fig. 11.4b. By logic simplification, the original Boolean function reduced to the normal function $d_1 d_2$, leaving the triggering input t_1, t_2 redundant.

VeriTrust can be considered as an "unused input identification" technique by looking for redundant inputs after setting all un-activated entries in verification tests to be don't-cares.

The overall framework of *VeriTrust* is shown in Fig. 11.5 which contains two parts: *tracer* and *checker*. The tracer traces verification tests to identify those signals that contain un-activated entries by tracing mechanisms. Then the checker analyzes these signals and determines whether any of them indeed contain redundant inputs and hence are potentially affected by HTs.

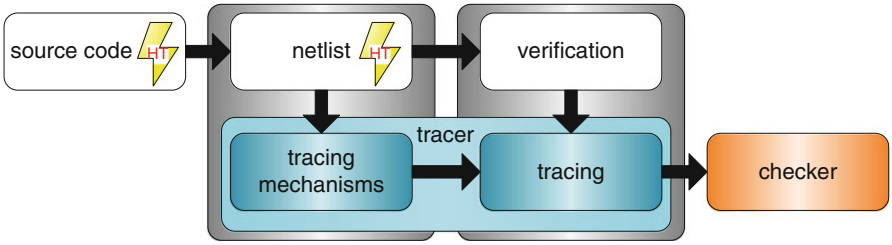


Fig. 11.5 The overview of *VeriTrust*

Tracer

Consider a particular signal whose fan-in logic cone may contain a HT, the responsibility of the tracer is to find out whether it contains any un-activated entries after verification tests. A straightforward method is to record the activation history of each and every entry, but it would require unaffordable memory space for large circuits and incur high runtime overhead. To resolve this problem, instead of tracing the activation history of each and every logic entry, a much more compressed tracing mechanism is proposed.

Before discussing the details, it is necessary to revisit some basics of Boolean functions. In general, any combinational circuit can be represented in the form of sum-of-products (SOP) and product-of-sums (POS). SOP uses OR operation to combine those on-set minterms, while POS uses AND operation to combine those off-set maxterms. Two minterms (maxterms) are *adjacent* if they have only one different literal.

Next, Zhang et al. introduced three new terms, *malicious on-set minterm*, *malicious off-set maxterm*, and *dummy term* that compose malicious function as follows.

Definition 1. The **malicious on-set minterm** is the on-set minterm in the malicious function whose adjacent minterms in the normal function are off-set.

Definition 2. The **malicious off-set maxterm** is the off-set maxterm in the malicious function whose adjacent maxterms in the normal function are on-set.

Definition 3. The **dummy term** is the on-set minterm or off-set maxterm in the malicious function whose adjacent minterms or maxterms in the normal function are also on-set or off-set.

With the above definitions, only the malicious on-set minterms and malicious off-set maxterms have malicious behavior. *Consequently, it is not necessary to set dummy terms as don't-cares to identify dedicated trigger inputs.* Figure 11.6 shows the K-Maps of two example HT-infected circuits (t_1 and t_2 are trigger inputs) to illustrate the above terms. The entry filled with vertical lines is malicious on-set minterm, as its neighboring entries in the normal function are all logic "0"s. The entry filled with horizontal lines is malicious off-set maxterm, as its neighboring

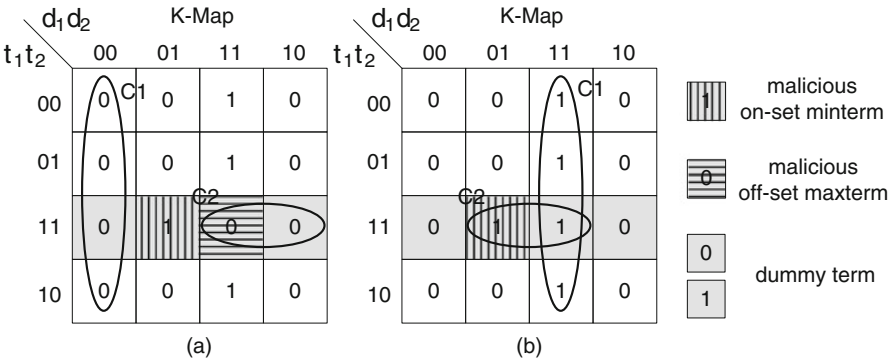


Fig. 11.6 Two HT-affected circuits triggered by $\{t_1, t_2\} = \{1, 1\}$

entries in the normal function are all logic "1"s. The remaining entries without vertical lines and horizontal lines in the malicious function are dummy terms, as they have the same values with their neighboring entries in the normal function. From the above definitions, when simplifying the circuit into the minimal form of SOP (POS), following observations have been obtained:

- Malicious on-set minterms and malicious off-set maxterms can only be combined with terms in the malicious function. This is because all the adjacent minterms (maxterms) of malicious on-set minterms (malicious off-set maxterms) in the normal function are off-set (on-set). For example, in Fig. 11.6a, one malicious off-set maxterm is combined with one dummy term (circled by C2), and in Fig. 11.6b, one malicious on-set maxterm is combined with one dummy term (circled by C2).
- Dummy terms can be combined with terms in the normal function as well as terms in the malicious function. For example, the circle C1 in Fig. 11.6a, b shows that the dummy term is combined with terms in the normal function; the circle C2 in Fig. 11.6a, b shows that the dummy term is combined with terms in the malicious function.

From the above, simplified products or sums containing malicious on-set minterms or malicious off-set maxterms cannot be activated during the verification. In other words, during the tracing process, we can record the activation history of products and sums instead of that of each logic entry, and hence the memory requirement and runtime overhead of our tracer can be dramatically reduced. The simplified products and sums can be obtained by simplifying the circuit to the minimum SOP and POS form, by leveraging the capability of logic synthesis tool.

The tracing procedure might still be time-consuming if the number of products and sums to be traced is large. Tracing overhead can be further reduced by periodically removing those activated sums and products and those un-activated ones whose signal is determined to be HT-free with the checker.

The tracer outputs a number of signals that have un-activated products or sums after applying verification tests. The checker then checks whether a particular signal is driven by any redundant input by assigning the corresponding un-activated products and sums to be don't-cares. If it is, it would be a suspicious HT-affected signal; otherwise it is guaranteed to be HT-free.

Checker

To identify whether a signal is driven by redundant inputs could be time-consuming if its fan-in logic cone is large. To mitigate this problem, three optimization methods are adopted in sequence for redundant input identification, which differ in the checking capability and time complexity.

Checker 1 simply checks whether there is any redundant input by removing un-activated products/sums from the signal's SOP/POS representation. Take the circuit in Fig. 11.6a as an example. One SOP can be represented as: $f = d_1d_2 + t_1t_2d_2$. If we remove the un-activated $t_1t_2d_2$ from the SOP, the logic function becomes $f = d_1d_2$ with redundant inputs t_1 and t_2. This efficient checking mechanism is effective in many cases, but it cannot guarantee complete identification of redundant inputs. This is because, whether checker 1 can find out redundant inputs depends on the SOP/POS representation of the signal. For example, if the circuit in Fig. 11.6b is represented as $f = \bar{t}_1d_1d_2 + \bar{t}_2d_1d_2 + t_1t_2d_2$, then removing the un-activated $t_1t_2d_2$ cannot leave t_1 and t_2 redundant.

Checker 2 leverages logic synthesis to re-simplify the function by considering un-activated products and sums as don't-cares. If an input does not appear in the synthesized circuit, it is a redundant input. This method is able to find most redundant inputs, but still cannot guarantee to find all since synthesis tool cannot guarantee optimality by employing heuristic algorithm for logic minimization.

Checker 3 verifies all inputs that are used in the un-activated products/sums one by one. If the change of an input would not cause the change of the function in all input patterns under the condition that un-activated products and sums are set as don't-cares, it should be a redundant input. Checker 3 can guarantee to find out all redundant inputs, but it is more time-consuming than Checker 1 and Checker 2.

To ensure complete identification capability while keeping computational time low, checker 1, checker 2, and checker 3 are run in a consecutive manner. For each signal examined, if a more efficient checker (e.g., checker 1) finds out a redundant input for a particular signal with un-activated products/sums, the products/sums are marked as a suspicious HT-affected signal. Otherwise, next checker is used for redundant input identification. If all the three checkers cannot find redundant inputs for the signal of interest, it is guaranteed to be HT-free. Eventually, the checker returns a list of suspicious signals that are potentially affected by HTs.

Algorithm 11.1: Suspicious wires detection in FANCI

```
1  for all modules m do
2      for all gates g in m do
3          for all output wires w of g do
4              T ← TruthTable(FanInCone (w));
5              V ← Empty vector of control values;
6              for all columns c in T do
7                  Compute control value of c;
8                  Add control(c) to vector V;
9              end for
10             Compute heuristics for V;
11             Decide w as suspicious or not;
12         end for
13     end for
14 end for
```

11.3.3.3 FANCI

FANCI [14], which stands for Functional Analysis for Nearly unused Circuit Identification, is a kind of *static* Boolean function analysis to find signals with weakly affecting inputs. It is based on the observation that a HT trigger input generally has a *weak* impact on output signals. Thus these weakly affecting inputs are probably inserted by adversaries as backdoor triggers. The primary goal of FANCI is to identify the "weakly affecting" dependency between inputs and outputs via a metric called *control value*, which measure the degree of control that an input has on the outputs of a digital circuit.

The overall detection algorithm can be seen in Algorithm 11.1. FANCI examines outputs of every gate in all modules inside the hardware design and constructs functional truth table from its fan-in cone. For each input feeding into the examined output, FANCI determines whether the input is suspicious by *control value* computed from truth table. Given an input wire w_i and output wire w_o, FANCI algorithm traverses all rows in the truth table of w_o, denoted by T, and count rows in T that w_o changed when flipping the value of w_i. The *control value* of w_i on w_o is defined as

$$\mathrm{CV}(w_i, w_o) = \frac{count}{size(T)} \tag{11.1}$$

where *count* denotes the total number of rows the change of w_i affecting the value of w_o while $size(T)$ denotes the size of truth table (i.e., the total number of rows in T).

For example, shown in Fig. 11.3a, which function can be denoted as $f = d_1d_2 + t_1t_2d_2$. Its truth table is shown in Table 11.2. There are only two rows, highlighted in gray, under which flipping t_1 leads to the change of the output, and hence the *control value* of t_1 on f is $\mathrm{CV}(t_1, f) = 2/2^4 = 0.125$. In the similar manner, a vector of *control values* $\mathbf{V}$ is acquired for f as $\mathbf{V} = [0.125, 0.125, 0.375, 0.625]$ containing the control values for input t_1, t_2, d_1, d_2. Since the sizes of truth tables

Table 11.2 Truth table of circuit $f = d_1d_2 + t_1t_2d_2$

t_1	t_2	d_1	d_2	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

grow exponentially with respect to the number of input wires, computing *control values* deterministically is exponentially hard. Necessary optimization is required for efficient and practical detection. They proposed to compute approximate *control value* by selecting a subset of rows from the complete truth table at uniformly random. For instance, N cases are chosen randomly for computing and for each case w_i is flipped and the total number of times that w_o changes is recorded, denoted by n. The approximate *control value* is $\frac{n}{N}$. It shall be noted that rows in truth table are selected uniformly at **random** to prevent potential adversaries from exploiting the sampling procedure and designing HTs evade FANCI.

When the vector of control values is computed for a signal under analysis, it is not trivial to determine whether it contains a HT trigger. Having only one weakly affecting input or a input that is only borderline weakly affecting is not sufficiently suspicious as it could simply be an inefficient implementation. FANCI includes the several heuristics, such as median and mean, to decide whether the signal is suspicious by considering all elements in the CV vector.

Median If an output is affected by HT triggers, the median is often close to zero. If the distribution in CV vector is irregular, the median may bring in unnecessary false positives.

Mean This metric is similar to median, but slightly sensitive to outliers. It is more effective when there are few unaffecting dependencies.

Median and Mean Consider the signals with both extreme median and mean values to mitigate the limitations and diminish false positives.

Triviality This metric calculates a weighted average of the vector of control values. Each weight of the wires in CV vector is measured by how often they are the only wire influencing the output to determine how much an output is influenced overall by its input. Suppose an output wire with $w_o = w_{i_1} \oplus \ldots \oplus w_{i_n}$ is analyzed by FANCI, all input wires have the control value of 1.0, but it is never the case that one input completely controls the output. Thus the triviality, i.e., the weight of every input is 0.5.

Table 11.3 HT detection with hardware functional verification and trust verification

	Static/dynamic	Detection method	Runtime
Functional verification	Dynamic	Activate the HT	Good
UCI by code coverage	Dynamic	Identify uncovered parts	Good
UCI by Hicks et al. [10]	Dynamic	Identify equal signal pair	Fair
VeriTrust [13]	Dynamic	Identify HT trigger inputs	Fair
FANCI [14]	Static	Identify weakly affecting inputs	Fair

	False negatives	False positive
Functional verification	HTs with rare trigger condition	None
UCI by code coverage	HTs in [2]	Few with thorough verification
UCI by Hicks et al. [10]	HTs in [2, 11]	Some with thorough verification
VeriTrust [13]	Unknown	Some with thorough verification
FANCI [14]	Possible with low threshold	Many with high threshold

For each metric, it is necessary to have a cut-off threshold for what is suspicious and what is not. The value is chosen either a priori or after examining the distribution of computed values.

11.3.3.4 Discussion

Table 11.3 summarizes the characteristics of existing solutions for HT detection. Since dynamic trust verification techniques (i.e., UCI and VeriTrust) analyze the corner cases of functional verification for HT detection, these two types of verification techniques somehow complement each other. Generally speaking, with more FV tests applied, the possibility for HTs being activated is higher while the number of suspicious circuitries reported by UCI and *VeriTrust* would decrease. As a static solution that does not depend on verification, one unique advantage of FANCI over the other solutions is that it does not require a trustworthy verification team. On the other hand, however, adversaries could also validate their HT designs using FANCI without necessarily speculating the unknown test cases used to catch them.

All trust verification techniques try to eliminate *false negatives* (a false negative would mean a HT that is not detected) whilst keeping the number of *false positives* as few as possible in order not to waste too much effort on examining benign circuitries that are deemed as suspicious. However, their detection capability is related to some user-specified parameters and inputs during trust verification. For example, FANCI defines a *cut-off threshold* for what is suspicious and what is not during Boolean functional analysis. If this value is set to be quite large, it is likely to catch HT-related wires together with a large number of benign wires. This, however, is a serious burden for security engineers because they have to evaluate all suspicious wires by code inspection and/or extensive simulations. If this value is set to be quite small, on the contrary, it is likely to miss some HT-related wires. Similarly, if only a

small number of FV tests are applied, UCI and *VeriTrust* would flag a large number of suspicious wires (all wires in the extreme case when no FV tests are applied), which may contain HT-related signals but the large amount of false positives make the following examination procedure infeasible.

11.4 Stealthy HT Designs Defeating Trust Verification

HT design and HT identification techniques are like arms race, wherein designers update security measures to protect their system while attackers respond with more tricky HTs. With the state-of-the-art hardware trust verification techniques such as UCI, *VeriTrust*, and FANCI being able to effectively identify existing HTs, no doubt to say, adversaries would adjust their tactics of attacks accordingly and *it has been revealed that new types of HTs can be designed to defeat these hardware trust verification techniques.*

11.4.1 HTs Evade UCI

UCI tracks signal pairs whose values remain the same during the verification process. The circuits between these signals are regarded as "unused circuit" and are potential HT candidates. The obvious way to evade UCI is to make all dependable signal pairs in HT differ at least once under non-trigger condition.

11.4.1.1 Motivational Case

Sturton et al. [11] showed a simple design consisting only of two multiplexers evades UCI. The design is illustrated in Fig. 11.7. This circuit evades detection by UCI because there is no dependent signal pairs that are always equal under non-trigger condition. There are two non-trigger inputs d_1, d_2 and two trigger inputs t_1, t_2, with trigger condition $\{t_1, t_2\} = \{1, 1\}$. The output function $f = d_0 d_1 + t_0 t_1 \bar{d}_1$ and its truth table are shown in Table 11.4. Under trigger condition, the malicious function $f = d_0 \bar{d}_1$ is showed while under non-trigger condition, the normal function $f = d_0 d_1$ is facilitated.

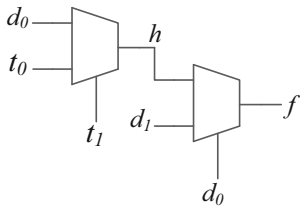


Fig. 11.7 Example defeating UCI

Table 11.4 Truth table of circuits in Fig. 11.7

	t_0	t_1	d_0	d_1	h	f	Comments
1	0	0	0	0	0	0	
2	0	0	0	1	0	0	$f \neq d_1$
3	0	0	1	0	1	0	$h \neq t_0, t_1$
4	0	0	1	1	1	1	$f \neq t_0, t_1$
5	0	1	0	0	0	0	
6	0	1	0	1	0	0	
7	0	1	1	0	0	0	$h \neq d_0$
8	0	1	1	1	0	1	$f \neq h$
9	1	0	0	0	0	0	
10	1	0	0	1	0	0	
11	1	0	1	0	1	0	$f \neq d_0$
12	1	0	1	1	1	1	
13	1	1	0	0	1	1	Trigger
14	1	1	0	1	1	1	Trigger
15	1	1	1	0	1	0	Trigger
16	1	1	1	1	1	1	Trigger

In Table 11.4, the rows in gray are the signal values under triggering condition, thus it cannot be seen during the verification tests. The final column notes when an input makes a dependent pair of signals unequal. If the verification test includes non-triggering patterns in row 2, 3, 4, 7, 8, 11, UCI would not flag any part of the circuit as malicious. Since the trigger condition for all of these inputs is false and the normal function is $d_0 d_1$, this circuits behaves identically with an AND gate in all test cases.

A rogue designer may utilize this fact and replace any AND gate with the malicious circuit in Fig. 11.7 and can hide from UCI detection. If this affects some security critical element of hardware, a hidden vulnerability is introduced by this HT.

The reason why UCI missed this behavior is that there is no intermediate function equivalent to the output function so that UCI cannot find a short-circuit with the output. The key idea to defeat UCI is to create a non-trigger function that is *not* equivalent to any internal functions.

11.4.2 HT Design Against UCI

Sturton et al. proposed to construct malicious Trojan in an incremental manner. They first built a set of small circuits that evade UCI and construct more stealthy HTs by combing these small circuits with basic circuit-building elements (e.g., AND gate, OR gate, MUX, etc.) via exhaustive search. Every newly created circuit is verified by UCI to determine whether it can evade UCI.

The intuition behind this incremental search is based on the fact every subcircuit of a UCI-evading circuit also evades UCI. This is simple to understand by considering the definition of "unused circuit." If a circuits evades UCI, meaning that every internal signal inside this circuits does not always equal to the signals in its fan-in cone. Thus the circuits between each internal signal and its primary inputs are a subcircuit and it evades UCI.

Before presenting the HT design algorithm, Sturton et al. defined several concepts for the ease of discussion:

Definition 4. An *admissible circuit* is a circuit that (1) contains exactly one trigger condition, (2) has at least one trigger inputs, (3) its output is independent from trigger input under non-trigger condition.

Definition 5. A circuit is *obviously malicious* if there exist two valuations to the inputs of the circuit, $x = (d, t)$, $x' = (d, t')$, in which t is a non-trigger condition while t' is trigger condition and the output of the circuit is different under these two conditions.

Definition 6. A circuit is a *stealthy circuit* if it evades UCI, i.e., there is no pair of dependent signals always equal in verification tests.

These three definitions perfectly characterize the three aspects of HTs that evades UCI. Firstly, HT shall contain a malicious function that can be triggered in specific conditions, i.e., obviously malicious. Second, HT shall pass the design-time verification, the output has nothing to do with trigger inputs, i.e., admissible. Finally, it cannot contain any signals that would be flagged as suspicious by UCI. The goal of HT construction algorithm is to find the class of HTs that contains above mentioned characters.

Algorithm 11.2 shows the proposed way to generate HTs from scratch. HT designers first shall fix the number of input signals to the circuit and the size of the circuit. Also, the basic blocks for building the circuit shall be determined. The construction algorithm maintains a workqueue of newly formed circuits which would serve as building blocks for future circuits. The workqueue initially contains the circuits that contains only one input.

Roughly speaking, in each iteration of the algorithm, one circuit is removed out of the workqueue and consider all ways to expand it to make a new stealthy circuit. If a new circuit is found satisfying three definitions, it is added to the database of all constructed HTs. The algorithm can be run multiple times with different parameters (e.g., number of inputs, size of circuit, basic building gates, etc.), and more versatile HTs would be added to the database.

After the searching algorithm is finished, a database containing circuits with various normal functions and malicious functions is formed. Rogue designers can replace arbitrary circuit in the normal design with HT-infected circuit with the same normal function by searching the database.

Algorithm 11.2: Construct circuits to defeat UCI

```
   // Initial Circuit, one for each input
 1 C_0 = (d_0);C_1 = (d_1);...;C_n = (t_m);
   // Set of circuits found that evades UCI
 2 completed_circuits = ∅;
   // Set of circuits used as building block for larger circuits
 3 workqueue = {C_0, C_1, ..., C_n};
   // Set of basic gates in building circuits
 4 gate_basis = {AND, OR, NOT, NAND, 2-input MUX};
 5 while length(workqueue) > 0 do
 6     curr_circuit = workqueue.pop();
 7     if curr_circuit is stealthy, admissible and oblivious malicious then
 8         Print curr_circuit;
 9     else
10         for all gate in gate_basis do
11             for all circ in completed_circuits ∪ {curr_circuit} do
12                 new_circuit = gate(curr_circuit, circ);
13                 if new_circuit is stealthy and not in completed_circuits then
14                     workqueue.append(new_circuit);
15                 end if
16             end for
17         end for
18         completed_circuits.add(curr_circuit);
19     end if
20 end while
```

11.4.3 HTs Evade VeriTrust

As discussed earlier, *VeriTrust* flags suspicious HT trigger inputs by identifying those inputs that are redundant under verification. Consequently, as it is shown in [15], the key idea to defeat *VeriTrust* is to make HT-affected signals driven by non-redundant inputs only under non-trigger condition.

11.4.3.1 Motivational Case

For any input that is not redundant under non-trigger condition, the following lemma must be true.

Lemma 1. *Consider a HT-affected signal whose Boolean function is* $f(h_1, h_2, \ldots, h_k)$. *Any input,* h_i, *is not redundant under non-trigger condition, as long as the normal function, denoted by* f_n, *cannot be completely represented without* h_i.

Proof. Since f_n cannot be completely represented without h_i, there must exist at least one pattern for all inputs except h_i under which $f_n(h_i = 0) \neq f_n(h_i = 1)$. Therefore, h_i is not redundant.

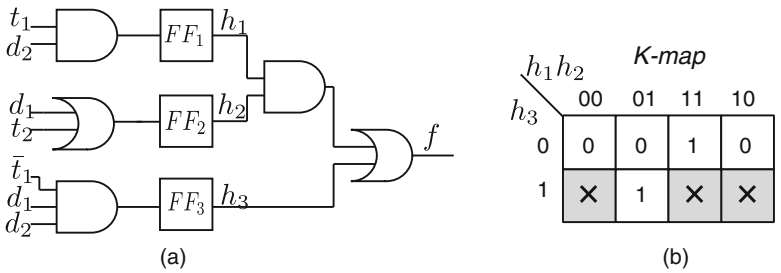


Fig. 11.8 Motivational example for defeating VeriTrust (**a**) HT-infected circuit (**b**) K-map

Inspired by Lemma 1, Fig. 11.8a shows a HT-infected circuit that is revised according to the circuit shown in Fig. 11.4, wherein the HT is activated when $\{t_1, t_2\} = \{1, 1\}$. In this implementation, the malicious product $t_1t_2d_2$ is combined with the product $t_1d_1d_2$ from the normal function and hidden in the fan-in cones of h_1 and h_2, where $h_1 = t_1d_2$ and $h_2 = d_1 + t_2$. The K-map of f is shown in Fig. 11.8b, where entries that cannot be activated under non-trigger condition are marked as "don't-cares." For this circuit, VeriTrust, focusing on the combinational logic, would verify four signals, f, h_1, h_2, and h_3. According to the K-map shown in Fig. 11.8b, it is clear that h_1, h_2, and h_3 are not redundant for f under non-trigger condition. Moreover, h_1, h_2, and h_3, which are not HT-affected signals, have no redundant inputs as well, since all of their input patterns can be activated under non-trigger condition. Therefore, the HT in Fig. 11.8a is able to evade VeriTrust.

By examining the implementation of this motivational case, it is clear that the mixed design of the trigger and the original circuit makes trigger condition for the HT-affected signal not visible for VeriTrust. In order to differentiate existing HTs and HTs like the one shown in Fig. 11.8a, two new terms are introduced in [15]: the *explicitly triggered HT* and the *implicitly triggered HT* as follows.

Definition 7. A HT is **explicitly triggered** if in the HT-affected signal's fan-in logic cone, there exists an input pattern that uniquely represents the trigger condition.

Definition 8. A HT is **implicitly triggered** if in the HT-affected signal's fan-in logic cone, there does not exist any input pattern that uniquely represents the trigger condition.

All explicitly-triggered HTs can be detected by VeriTrust, as they contain dedicated trigger inputs which can be identified by VeriTrust. The HT shown in Fig. 11.8a is implicitly-triggered, since the trigger condition is hidden in the h_1h_2 which also contains certain circuit's normal functionalities.

The HT design method shown in next section is motivated by the above example and observations by implementing implicitly-triggered HTs to defeat VeriTrust.

11.4.3.2 HT Design Against VeriTrust

As *VeriTrust* only focused on detecting HT trigger inputs in the combinational logic block wherein the output is HT-affected signal. The HT design methodology evading *VeriTrust* also focused on the combinational logic. According to Lemma 1, the approach to defeat *VeriTrust* implements the implicitly triggered HT with the following two steps:

- Combine all malicious on-set terms[1] with on-set terms from normal function, and re-allocate sequential elements (e.g., flip-flops) to hide the trigger in multiple combinational logic blocks.
- Simplify all remaining on-set terms and re-allocate sequential elements if they contain trigger inputs.

Note that, only on-set terms are selected, since the circuit can be explicitly represented by the sum of all on-set terms.

The defeating method against *VeriTrust* can be further illustrated as follows. Consider a circuit with an explicitly triggered HT, and its Boolean function can be represented by

$$f = \sum_{c_{n_i} \forall C_n,\, p_{n_j} \forall P_n,} c_{n_i} p_{n_j} + \sum_{c_{m_i} \forall C_m,\, p_{m_j} \forall P_m} c_{m_i} p_{m_j}, \tag{11.2}$$

where P_n and P_m driven by functional inputs denote the set of all patterns that make the normal function and malicious function output logic "1", while C_n and C_m driven by trigger inputs denote the set of non-trigger conditions and the trigger conditions, respectively.

For the sake of simplicity, the analysis began with the case where the malicious function contains only one malicious on-set term, $c_{m_0} p_{m_0}$. Suppose $c_{n_0} p_{n_0}$ from the normal function is selected to combine with $c_{m_0} p_{m_0}$. Let f'_n be all the on-set terms from the normal function except $c_{n_0} p_{n_0}$. Then, f can be given by

$$f = f'_n + (c_{n_0} p_{n_0} + c_{m_0} p_{m_0}). \tag{11.3}$$

Suppose $c_{n_0} p_{n_0}$ and $c_{m_0} p_{m_0}$ have the common literals, $c^c p^c$, and then

$$f = f'_n + c^c p^c (c^r_{n_0} p^r_{n_0} + c^r_{m_0} p^r_{m_0}), \tag{11.4}$$

where

$$\begin{aligned} c_{n_0} &= c^c c^r_{n_0}; \quad & p_{n_0} &= p^c p^r_{n_0}; \\ c_{m_0} &= c^c c^r_{m_0}; \quad & p_{m_0} &= p^c p^r_{m_0}. \end{aligned} \tag{11.5}$$

[1] Malicious on-set term is the on-set term in the malicious function whose adjacent terms in the normal function are off-set [13]. On-set term and off-set term are terms that make the function output logic "1" and logic "0", respectively.

After that, the circuit is re-synthesized and the flip-flops are re-allocated to make f become

$$f = h_1 h_2 + h_3, \tag{11.6}$$

where

$$\begin{aligned} h_1 &= c^c p^c \\ h_2 &= c^r_{n_0} p^r_{n_0} + c^r_{m_0} p^r_{m_0}. \\ h_3 &= f'_n \end{aligned} \tag{11.7}$$

h_1, h_2, and h_3 are outputs of the re-allocated flip-flops.

As can be observed in Eqs. (11.6) and (11.7), the key of the defeating method is to extract common literals from the malicious on-set term and the on-set term from the normal function and hide the trigger into different combinational logic. With the above, f, h_1, h_2, and h_3 would have no redundant inputs under non-trigger condition.

For multiple malicious on-set terms, the above method can also be used to combine each of them with one on-set term from the normal function and then hide the trigger in different combinational logic blocks. Finally, the output function f can be represented as:

$$f = \sum_{i=0}^{k-1} (h_{2i+1} h_{2i+2}) + h_{2k+1}, \tag{11.8}$$

where

$$\begin{aligned} h_{2i+1} &= c^{c_i} p^{c_i} \\ h_{2i+2} &= c^{r_i}_{n_i} p^{r_i}_{n_i} + c^{r_i}_{m_i} p^{r_i}_{m_i}. \\ h_{2k+1} &= f'_n \end{aligned} \tag{11.9}$$

It is easy to prove that $h_1, h_2, \ldots, h_{2k+1}$ and f have no redundant inputs under non-trigger condition. Note that the HT can be spread over multiple sequential levels by further combining the trigger logic driving $h_1, h_2, \ldots, h_{2k+1}$ with normal logic.

The defeating approach shown above can defeat *VeriTrust* in theory if a complete verification is performed. However, in practice, due to stringent time limit and ever-increasing circuit complexity, it is very hard to guarantee the verification is enough for HT detection. Hence probability of HT trigger inputs being flagged as redundant inputs still exists because of insufficient verification. The authors proposed the following three optimizations:

- Combine simplified malicious products (rather than malicious on-set terms) with on-set terms from the normal function, so that fewer terms from the normal function are required to be activated to evade VeriTrust.

Algorithm 11.3: The flow to defeat VeriTrust

```
1 Simplify Boolean function of this combinational logic;
2 Conduct the simulation with tests guessed by attackers to obtain the probability of each
  product;
3 foreach simplified malicious product do
4 |   Greedily combine it with the product from the normal function with the largest
  |   activation probability and hide the trigger in different combinational logic blocks;
5 end foreach
6 Re-allocate flip-flops for the remaining products.
```

- Choose simplified products from the normal function to be combined with simplified malicious products, so that any of the terms in the product from the normal function being activated can make HT evade VeriTrust.
- Choose those simplified products from the normal function with high activation probabilities to be combined with malicious products. This method requires the knowledge about the probability of products from the normal function, which can be estimated by speculating on the test cases used in functional verification [2, 11].

The flow to defeat *VeriTrust* is illustrated in Algorithm 11.3. It first simplifies the Boolean function of the combinational logic of HT-affected signals, and then conducts simulation with speculated test cases to obtain the probability of each product. After that, a loop is used to hide HT triggers whenever possible. In each iteration, one malicious product is combined with one product from the normal function with the largest activation probability and it is hidden in different combinational logic blocks. At last, flip-flops are re-allocated for the remaining products.

11.4.4 HTs Evade FANCI

Since FANCI identifies signals with weakly affecting inputs within the combinational logic block, the key idea to defeat FANCI is to make the control values of all HT-related signals comparable to those of functional signals.

11.4.4.1 Motivational Case

The authors in [15] started with the case shown in Fig. 11.9 to illustrate the key idea of defeating FANCI. Figure 11.9a, b presents a regular multiplexer (MUX) and a malicious one with a rare trigger condition, respectively. FANCI is able to differentiate the two types of MUXes and flag the malicious one since the trigger inputs, denoted by $t_0, t_1, \ldots, t_{63}$, have very small control values for the output $O(\frac{1}{2^{65}})$.

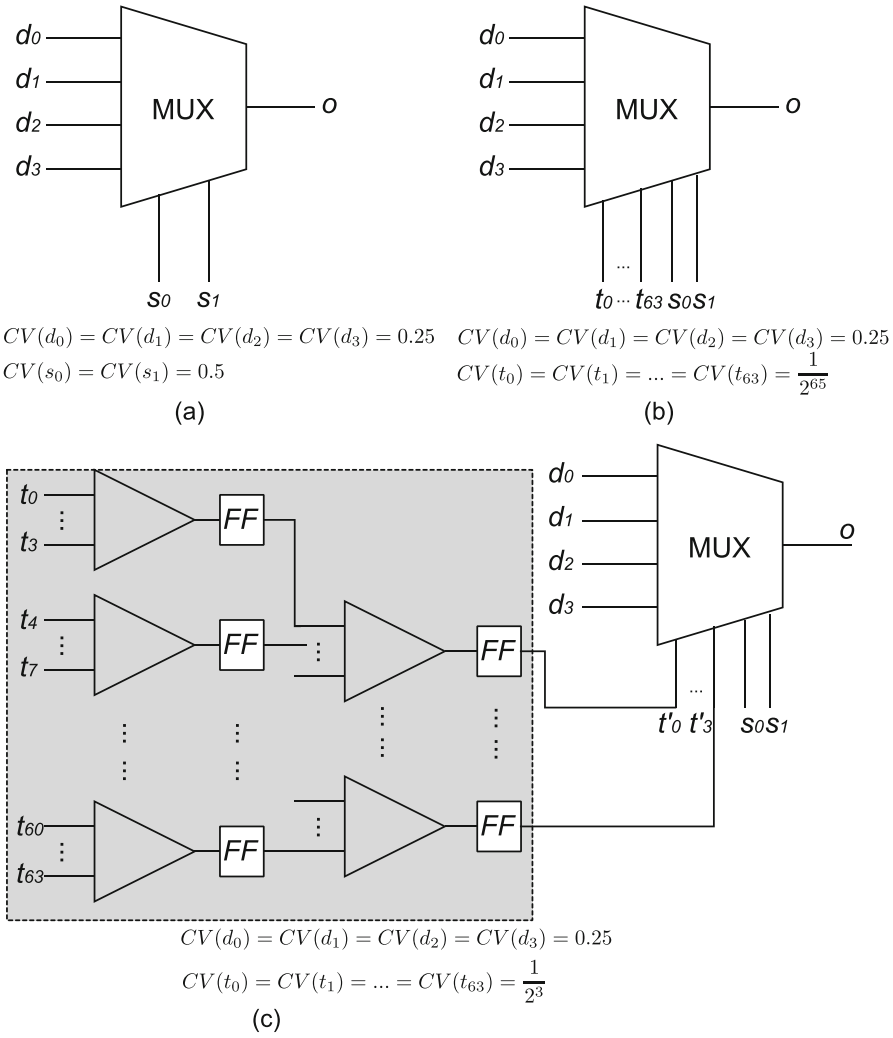


Fig. 11.9 Motivational example for defeating FANCI. (**a**) A standard MUX. (**b**) A malicious MUX. (**c**) A malicious MUX with modified implementation

From this example, we can see that the main reason for HT-related signals (e.g., *o* in Fig. 11.9b) having weakly affecting inputs is that it is driven by a number of trigger inputs in its fan-in combinational logic cone. Consider a signal driven by a combinational logic block with *m* trigger inputs and *n* functional inputs. The size of the truth table for this particular HT-related signal is given by:

$$size(T) = 2^{m+n}. \tag{11.10}$$

For any trigger input, denoted by t_i, those input patterns under which t_i influences the output should meet two requirements: (1) all trigger inputs other than t_i are driven by the trigger values[2]; (2) flipping t_i results in the change of the output value. There are in total 2^{n+1} input patterns meeting the first requirement. Among them, how many further satisfying the second requirement depends on the actual difference between the malicious function and the normal function, because they may output the same value under certain functional inputs. At the same time, they cannot always output the same value because otherwise there would be no malicious behavior. Therefore, the number of input patterns satisfying both requirements is bounded at:

$$2^1 \leq counter \leq 2^{n+1}. \tag{11.11}$$

With Eqs. 11.10 and 11.11, the control value of t_i on the corresponding HT-related signal is bounded at:

$$\frac{1}{2^{n+m-1}} \leq \mathrm{CV}(t_i) = \frac{counter}{size(T)} \leq \frac{1}{2^{m-1}}. \tag{11.12}$$

In order to make FANCI difficult to differentiate HT-related signals and function signals, the control values of HT-related signals shall be comparable to those of functional signals. As indicated by Eq. (11.12), reducing m has an exponential impact on the increase of control values. Thus, the authors modify the implementation of the malicious MUX by balancing these trigger inputs into multiple sequential levels (see Fig. 11.9c). In this way, the number of trigger inputs is controlled to be no more than four for any combinational block, rendering the control value of each trigger input comparable with those of functional inputs.

Motivated by the above, the approach of defeating FANCI is to reduce the number of trigger inputs in all the combinational logic blocks that drive HT-related signals, and it can be achieved by spreading HT trigger inputs among multiple sequential levels.

11.4.5 HT Design Against FANCI

In this section, two kinds of HTs are considered. Typically, the trigger of a stealthy HT consists of both combinational part (❶ in Fig. 11.10) and a sequential part (❷ in Fig. 11.10). The defeating algorithm for FANCI need to deal with them separately because different methods are adopted to handle the extra delay induced by additional sequential levels. Algorithm 11.4 presents the flow to defeat FANCI.

For the combinational logic blocks in ❶, the defeating method is similar to the one shown in Fig. 11.9c. As can be seen, the original trigger combinational logic

[2]Trigger values are logic values for trigger inputs to satisfy trigger condition.

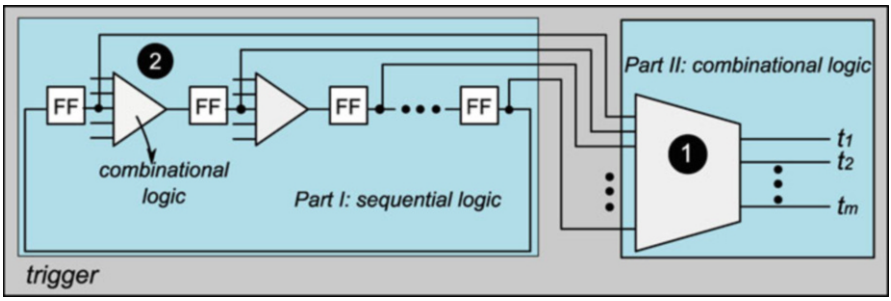


Fig. 11.10 Trigger design for a typical HT

Algorithm 11.4: The flow to defeat FANCI

```
1  N_T = 2;
2  do
3  |  DefeatFANCI(N_T++);
4  while (The hardware cost is larger than a given constraint.);
   /* One step to defeat FANCI                                   */
5  DefeatFANCI(N_T)
   |  /* For combinational logic of ❶                           */
6  |  foreach The fan-in cone of the input of the flip-flop do
7  |  |  if the number of trigger inputs > N_T then
8  |  |  |  Balance the trigger in the multiple sequential levels;
9  |  |  end if
10 |  end foreach
   |  /* For combinational logic of ❷                           */
11 |  Find out the maximum number of trigger signals, denoted by N_max, within a
   |  combinational logic cone;
12 |  if N_max > N_T then
13 |  |  Introduce multiple small FSMs until N_max ≤ N_T;
14 |  end if
15 end
```

with a large number of trigger inputs is spread among multiple combinational logic blocks, such that the number of trigger inputs in each combinational logic block is no more than N_T (a value used to tradeoff HT stealthiness and hardware overhead). As shown in Algorithm 11.4 (Lines 6–10), only the inputs of flip-flops are examined, rather than all signals. This is because, as long as the number of trigger inputs for the input of every flip-flop is smaller than N_T, the number of trigger inputs for every internal signal is also smaller than N_T.

The sequential part of a HT trigger can be represented by a finite-state machine (FSM) and the trigger inputs are used to control the state transitions (see Fig. 11.11a). For the combinational logic blocks in ❷ that sits inside the FSM, the above defeating method is not applicable because it introduced extra sequential levels. The additional pipeline delay incurred would change trigger condition.

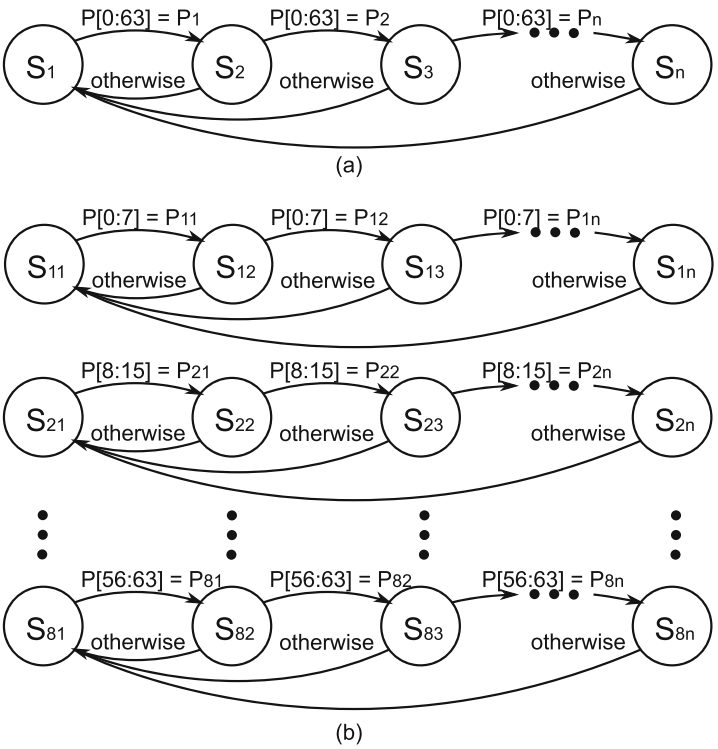


Fig. 11.11 The proposed sequential trigger design to defeat FANCI. (**a**) The original FSM. (**b**) The multiple small FSMs

Instead, the original FSM is partitioned into multiple small FSMs, e.g., as shown in Fig. 11.11b, the FSM with 64 trigger inputs is partitioned into eight small FSMs. By doing so, the number of trigger inputs in each small FSM is reduced to eight for this example, and it can be further reduced by introducing more FSMs. The HT is triggered when all the small FSMs reach certain states simultaneously.

Note that the proposed defeating method against FANCI has no impact on both circuit's normal functionalities and HT's malicious behavior, because this method only manipulates the HT trigger design, which is separated from the original circuit and the HT payload.

As the stealthiness of a HT is mainly determined by the number of trigger inputs in each combinational logic, this is achieved by finding the value of N_T in a greedy manner. That is, as shown in Algorithm 11.1, we start with $N_T = 2$ and gradually increase it until the cost of applying the defeating method is lower than the given constraint.

11.4.6 Discussion

The defeating approach against FANCI and that against *VeriTrust* do not interfere with each other. On the one hand, defeating algorithm for FANCI focuses on reducing the number of trigger inputs in the combinational logic blocks used in HT triggers without changing their logic functions; on the other hand, defeating method for *VeriTrust* implements the implicitly triggered HT without increasing the number of trigger inputs in any combinational logic.

Moreover, the HTs designed for defeating FANCI and *VeriTrust* would not influence the stealthiness of HT designs in Sect. 11.4.1.1 against UCI. Firstly, these methods do not change HT trigger condition and hence it has no impact on functional verification. For the UCI technique in [10], on the one hand, the signals introduced in algorithm for defeating FANCI are within the HT trigger unit driven by different trigger inputs and they are unlikely to be always equal during functional verification; on the other hand, *VeriTrust* defeating method combines parts of the normal functionalities with the HT trigger and hence is also unlikely to create equal signal pairs during verification.

With the above, all three HT design methodologies can be mixed in a single design to make the HT resistant to all known trust verification techniques while still passing functional verification.

References

1. S.T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, Y. Zhou, Designing and implementing malicious hardware, in *LEET*, vol. 8 (2008), pp. 1–8
2. J. Zhang, Q. Xu, On hardware Trojan design and implementation at register-transfer level, in *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2013), pp. 107–112
3. S. Skorobogatov, C. Woods, Breakthrough silicon scanning discovers backdoor in military chip, in *Proc. International Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2012), pp. 23–40
4. Y. Liu, Y. Jin, Y. Makris, Hardware Trojans in wireless cryptographic ICs: silicon demonstration & detection method evaluation, in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2013), pp. 399–404
5. M. Beaumont, B. Hopkins, T. Newby, *Hardware Trojans-Prevention, Detection, Countermeasures (A Literature Review)* (Australian Government Department of Defense, 2011)
6. Defense Science Board Task Force on High Performance Micorchip Supply, Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics (United States Department of Defense, 2005)
7. Y. Jin, N. Kupp, Y. Makris. Experiences in hardware trojan design and implementation, in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust* (2009), pp. 50–57
8. Trust-Hub Website, https://www.trust-hub.org/
9. S. Wei, K. Li, F. Koushanfar, M. Potkonjak, Hardware Trojan horse benchmark via optimal creation and placement of malicious circuitry, in *Proceedings of ACM/IEEE Design Automation Conference* (2012), pp. 90–95

10. M. Hicks, M. Finnicum, S.T. King, M.K. Martin, J.M. Smith, Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2010), pp. 159–172
11. C. Sturton, M. Hicks, D. Wagner, S. T King, Defeating UCI: building stealthy and malicious hardware, in *Proceedings of the IEEE International Symposium on Security and Privacy (SP)* (2011), pp. 64–77
12. J. Bormann, et al., Complete formal verification of TriCore2 and other processors, in *Design and Verification Conference* (2007)
13. J. Zhang, F. Yuan, L. Wei, Z. Sun, Q. Xu, VeriTrust: verification for hardware trust, in *Proc. IEEE/ACM Design Automation Conference (DAC)* (2013), pp. 1–8
14. A. Waksman, M. Suozzo, S. Sethumadhavan, FANCI: identification of stealthy malicious logic using boolean functional analysis, in *Proceedings of the ACM Conference on Computer and Communication Security (CCS)* (2013), pp. 697–708
15. J. Zhang, F. Yuan, Q. Xu, DeTrust: defeating hardware trust verification with stealthy implicitly-triggered hardware trojans, in *Proceedings of the ACM Conference on Computer and Communication Security (CCS)* (2014), pp. 153–166

Chapter 12
Verification and Trust for Unspecified IP Functionality

Nicole Fern and Kwang-Ting (Tim) Cheng

12.1 Introduction

Electronic devices and systems are now ubiquitous and influence the key aspects of modern life such as freedom of speech, privacy, finance, science, and art. Concern about the security and reliability of our electronic systems and infrastructure is at an all-time high. Securing electronic systems is extremely difficult because an attacker only needs to find and exploit a single weakness to perform malicious actions whereas defenders must protect the system against an infinite set of possible vulnerabilities to ensure it is secure.

Security techniques target detection/prevention of a class of vulnerability given a specific threat model. The task of enumerating and addressing all threat models and vulnerabilities is never complete, but this chapter contributes to this task by exploring a novel class of vulnerability: the opportunity in most hardware designs for an attacker to hide malicious behavior entirely within **unspecified functionality**.

12.1.1 Unspecified IP Functionality

Verification and testing is a major bottleneck in hardware design, and as design complexity increases, so does the productivity gap between design and verification [1]. It is estimated that over 70 % of hardware development resources are consumed

N. Fern (✉)
UC Santa Barbara, Santa Barbara, CA, USA
e-mail: nicole@ece.ucsb.edu

K.-T. (Tim) Cheng
Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
e-mail: timcheng@ust.hk

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_12

by the verification task. To address this challenge, commercial verification tools and academic research developed over the past several decades have focused on increasing confidence in the correctness of *specified functionality*. Important design behavior is modeled, then various tools and methods are used to analyze the implementation at various stages in the chip design process to ensure the implementation always matches the golden reference model.

Behavior which is not modeled will not be verified by existing methods, meaning any security vulnerabilities occurring within unspecified functionality will go unnoticed. In modern complex hardware designs, which now contain several billion transistors, there always exists unspecified functionality. It is simply impossible to enumerate what the desired state of several billion transistors or logic gates should be at every cycle when behavior depends not only on the internal state of the system, but also on external input from the environment the device is embedded in. It is only feasible to model and verify aspects of the design with functional importance.

Because behaviors at a good fraction of signals for many operational cycles are unspecified, an **attacker can modify this functionality with impunity without detection** by existing verification methods.

This chapter explores how an attacker can embed malicious behavior *completely* within unspecified functionality whereas most related research explores how to detect violations of specified behavior occurring under extremely rare conditions, where the main challenge is identifying these conditions.

12.1.2 Hardware Trojans

Malicious functionality inserted into a chip is called a *Hardware Trojan*. Hardware Trojans are a major concern for both semiconductor design houses and the US government due to the complexity of the chip design ecosystem [2, 3]. Economic factors dictate that the design, manufacturing, testing, and deployment of silicon chips are spread across many companies and countries with different and often conflicting goals and interests. If a single party involved deems it advantageous to insert a Hardware Trojan, the consequences can be catastrophic.

Goals of previously proposed Hardware Trojans range from denial-of-service attacks such as forcing premature circuit aging [4] and on-chip system bus deadlock [5] to subtler attacks which attempt to gain undetected privileged access on a system [6], leak secret information through power or timing side channels [7], or weaken random number generator output [8]. Many Trojan taxonomies exist [9–11], categorizing Trojans based on the design phase they are inserted, the

triggering mechanism, and malicious functionality accomplished (payload). Most existing Trojans can be divided into the following categories:

1. The logic functions of some design signals are altered, causing the circuit to violate the system specification
2. The Trojan leaks information through side-channels, and no functionality of any existing signals is modified

This chapter introduces and addresses a third, less studied type of Trojan:

3. The logic functions of **only** those design signals which have **unspecified behavior** are altered to add malicious functionality without violating system specifications

An example of a Class 3 Trojan is the malicious circuitry shown in red in Fig. 12.1. The behavior of the Trojan-infested circuit differs from the normal functionality of the FIFO only when the value of `read_data` is unspecified. The FIFO writes the data currently at the input `write_data` to the buffer when the signal `write_enable` is 1 and the FIFO is not full, and places data from the buffer on the `read_data` output when the signal `read_enable` is 1 and the FIFO is not empty. One could ask what should the value of `read_data` be when `read_enable` is 0 or the FIFO is empty?

It may seem logical to assume the value of `read_data` under such conditions retains its value from the previous valid read, but what if the FIFO has never been written to or read from before? In this case the value is unknowable, and cannot be specified. It is very likely the verification effort will only examine the value of `read_data` when `read_enable` is 1 and the FIFO is not empty because it is assumed that any circuitry in the fan-out of `read_data` is only used during a valid FIFO read. **The value of `read_data` when `read_enable` is 0 is unspecified.**

However, this means an attacker can modify the FIFO design to leak secret information on the `read_data` output during all cycles for which the unverified conditions hold. This malicious circuitry is shown in red in Fig. 12.1. It should

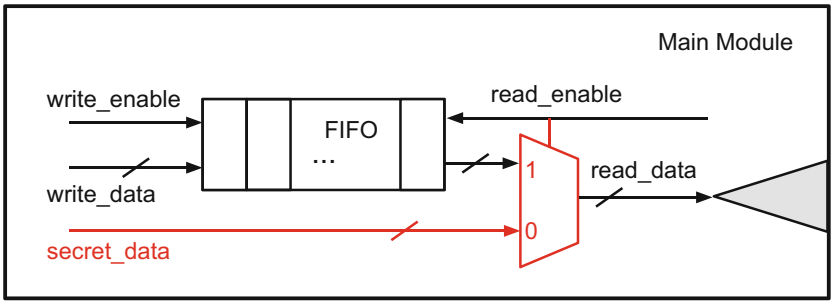


Fig. 12.1 FIFO infested with a Class 3 Trojan (Trojan circuitry affecting unspecified functionality to leak information shown in *red*)

be emphasized that these conditions occur quite frequently making this Trojan behavior hard to flag using existing pre-silicon detection methodologies relying on the identification of behavior occurring only under rare conditions.

Many of the Trojans proposed in literature hide from the verification effort using extremely rare triggering conditions (Class 1 Trojans). Examples of stealthy Trojan triggers are counters, which wait thousands of cycles before changing circuit functionality, or pattern recognizers, which wait for a "magic" value or sequence to appear on the system bus [12] or as plaintext input in cryptographic hardware [13–15]. Trojans with these triggering mechanisms generally deploy a payload which clearly violates system specifications (such as causing a fault during a cryptographic computation or locking the system bus).

Existing pre-silicon Trojan detection methods assume Trojans violate the design specification under carefully crafted rare triggering conditions, and focus on identifying the structure of this triggering circuitry in the RTL code or gate-level netlist [16–19]. These methods identify "almost unused" logic, where rareness is quantified by an occurrence probability threshold. This probability is either computed statically using approximate boolean function analysis [16, 17] or based on simulation traces [18, 19].

The Trojans addressed in this chapter do not rely on rare triggering conditions to stay hidden, but instead **only** alter the logic functions of design signals which have **unspecified behavior**, meaning the Trojan **never** violates the design specification. Addressing this Trojan type requires a different approach from both existing Trojan detection and hardware verification methods, since traditional verification and testing excludes analysis of unspecified functionality for efficiency, and focuses primarily on conformance checking. Identifying unspecified functionality that either contains a Hardware Trojan or could be exploited by an attacker in the future is difficult because by definition unspecified functionality is not modeled or known by the design/verification team.

The remainder of this chapter provides techniques to ensure unspecified design functionality is secure. Section 12.2 explores the most straightforward unspecified functionality to define: RTL don't cares. Several examples of how information can be leaked by only modifying don't care bits are given, then a design analysis method identifying the subset of don't care bits susceptible to Trojan modification is provided [20]. Section 12.3 presents a method to identify unspecified functionality beyond RTL don't cares (for example, the `read_data` signal in the FIFO example may not necessarily be assigned X's when `read_enable` is 0). The identification method is based on mutation testing and in a case study presented in Sect. 12.3 an entire class of Trojan modifying bus functionality only during idle cycles is discovered [21]. Section 12.4 expands upon this discovery and presents a general model for creating a covert Trojan communication channel between SoC components by altering existing on-chip bus signals only when they are unspecified and outlines how a Trojan channel can be inserted undetected in several widely used standard bus protocols such as AMBA AXI4 [22]. We then summarize our contributions and conclude in Sect. 12.5.

12.2 Trojans in RTL Don't Cares

Don't cares are a concept used to describe the set of input combinations to a boolean function for which the output can be either 0 or 1. For example, imagine the first 10 letters of the alphabet are to be displayed on a 7-segment display based on the binary value encoded using 4 switches. A logic function for each LED segment (there are seven in total) determines if the segment is ON or OFF for each of the ten combinations. However, 4 switches allow for 16 possible input combinations even though only 10 are required for the design. For six input combinations, the ON/OFF value of the LED segments does not matter given the problem specification.

If truth tables and Karnaugh maps for each segment are used to derive the gate-level representation for this toy design, the rows in the truth table corresponding to the 6 don't care input combinations can be assigned either 0 or 1 to minimize the overhead of the resulting logic. Modern designs are not specified using truth tables, but instead by Hardware Description Languages such as Verilog or VHDL. Both languages allow don't cares to be expressed, and the freedom provided by these don't cares allows the synthesis tool to optimize the resulting logic. In this chapter we focus on Verilog, but the concepts easily apply to VHDL. In Verilog, don't cares can be expressed using the literal "X" in the right-hand side of an assignment, meaning for the conditions under which the assignment statement executes, the variable being assigned "X" can be 0 or 1.

Unfortunately X's are also used to represent unknowns in addition to don't cares making the interpretation of X's confusing. X's appearing in RTL code have different semantics for simulation and synthesis. In RTL simulation, an X represents an unknown signal value, whereas in synthesis, an X represents a don't care, meaning the synthesis tool is free to assign the signal either 0 or 1.

During RTL simulation there are two possible sources of X's: (1) X's specified in the RTL code (either explicitly written by the designer or implicit such as a case statement with no default), and (2) X's resulting from uninitialized or un-driven signals, such as flip-flops lacking a known reset value or signals in a clock-gated block. X's from source 1 are don't cares, and are assigned values during synthesis, meaning they are *known* after synthesis, whereas X's from source 2 may be unknown until the operation of the actual silicon.

The Trojans we propose take advantage of source 1 X's, and clearly, if the design logic is fully specified, and don't cares never appear in the Verilog code, these Trojans cannot be inserted. However, don't cares have been used for several decades to minimize logic during synthesis [23], and forbidding their use can lead to unacceptable area/performance/power overhead. For the case study presented in Sect. 12.2.3, replacing all X's in the control unit Verilog with 0's results in almost an 8 % area increase for the block.

Turpin [24] and Piper and Vimjam [25] give an industry perspective and overview of the many problems caused by RTL X's during chip design/verification/debug along with a survey of existing techniques and tools which address X-issues. Simulation discrepancies between RTL and gate-level versions of a design due to

X-optimism and X-pessimism, and propagation of unknown values due to improper reset or power management sequences [26] are all issues addressed by existing research and commercial tools.

This section presents yet another issue resulting from the presence of X's in RTL code, and provides further incentive to allocate verification resources to these existing X-analysis tools. However, existing tools aim to uncover *accidental* functional X-bugs, while the Trojans we propose can be considered a special pathological class of X-bug specifically crafted with malicious intent to avoid detection during functional verification.

This means that X-analysis tools which focus only on providing RTL simulation with accurate X semantics, perform X-propagation analysis only for scenarios occurring during simulation-based verification, or formal methods which only analyze a limited number of cycles (ex. the reset sequence) do not adequately address the proposed threat. Through the examples in the remainder of the section we aim to highlight the aspects of this new threat that differ most from the existing X-bugs targeted by commercial and academic tools.

12.2.1 Illustrative Examples

Example 1 (Trojan Modifying Don't Care Bits to Leak Key). To illustrate how don't cares can be exploited to perform malicious functionality, a contrived example is presented for illustrative purposes. The module given in Listing 12.1 transforms a 4-bit input by either inverting, XORing with a secret key value, or passing the data to the output unmodified. The choice between the three transformations is selected using a 2-bit control signal, `control`. When `control=11`, Line 17 specifies that `tmp` can be assigned any value by the synthesis tool to minimize the logic used.

Listing 12.1 simple.v

```
1  module simple (clk,reset,control,data,key,out);
2  input clk, reset;
3  input [1:0] control;
4  input [3:0] data, key;
5  output reg [3:0] out;
6  reg [3:0] tmp;
7  //tmp only assigned a meaningful value
8  //if control signal is 00, 01 or 10
9  always @ (*) begin
10   case(control)
11     2'b00: tmp <= data;
12     2'b01: tmp <= data ^ key;
13     2'b10: tmp <= ~data;
14   //Trojan logic ------
15   //2'b11: tmp <= key;
16   //------------------
17     default: tmp <= 4'bxxxx;
18   endcase
```

```
19 end
20 always @ (posedge clk) begin
21   if (~reset) out <= 4'b0;
22   else out <= tmp;
23 end
24 endmodule
```

An attacker can take advantage of the implementation freedom given by the RTL by assigning `key` to `tmp`, causing the secret key value to appear at the output of this module. The Trojan can be inserted in the RTL code by uncommenting Line 15, or at gate-level by modifying the netlist after synthesis.

It should be emphasized that in either case, since the assignment of `tmp` during the `control=11` condition is **unspecified**, it is impossible to detect the Trojan even if the design can be exhaustively simulated, or a perfect equivalence checker can compare the golden and Trojan implementations. As an example, Cadence Conformal LEC [27] was used to perform two experiments: equivalence checking between Golden RTL and Trojan RTL, and equivalence checking between Golden RTL and a Trojan infested netlist. In both cases, the equivalence checker was unable to detect the presence of the Trojan functionality.

It should also be noted that the don't cares assigned to `tmp` in Line 17 are *useful* to the attacker because:

1. The don't care assignment is reachable
2. A primary output (which the attacker can observe) differs depending on the value of the don't care bits

Example 2 (Unreachable and Non-Propagating X's). In the previous example, all the don't care bits are dangerous and should be disambiguated in the RTL code. The following example (similar to Example 1 with the addition of a 3-bit FSM with five reachable states) illustrates that not all don't cares are dangerous, and that the goal of any Trojan prevention or X-analysis technique is to identify only the dangerous X's and allow the synthesis tool to use the remaining don't cares for logic minimization.

In Listing 12.2 there are six total assignments of 1-bit don't care values. One could replace these X's in the Verilog code with 6 1-bit signals, $dc_0, dc_1, \ldots, dc_5$. The attacker can then choose to assign other internal design signals (such as key bits) to the don't care bits or leave them for the synthesis tool to assign. Line 27 can be re-written as:

$$\texttt{default: pattern <= \{}dc_0\texttt{, }dc_1\texttt{, }dc_2\texttt{, }dc_3\texttt{\};}$$

Listing 12.2 simple_state.v

```
1  module simple_state(clk,reset,control,data,key,out);
2  input clk, reset;
3  input [1:0] control;
4  input [3:0] data, key;
5  output reg [3:0] out;
6  reg [3:0] tmp;
7  reg [2:0] counter, next_counter;
8  reg [3:0] pattern;
9  //Truncated Counter 0-4
10 //5,6, and 7 never appear
```

```
11 always @(*) begin
12   if (counter < 3'h4)
13      next_counter <= counter + 3'b1;
14   else next_counter <= 3'b0;
15 end
16 always @(posedge clk) begin
17   if (~reset) counter <= 3'b0;
18   else counter <= next_counter;
19 end
20 always @(*) begin
21   case(counter)
22     3'd0: pattern <= 4'b1010;
23     3'd1: pattern <= 4'b0101;
24     3'd2: pattern <= 4'b0011;
25     3'd3: pattern <= 4'b1100;
26     3'd4: pattern <= 4'b1xx1;
27     default: pattern <= 4'bxxxx;
28   endcase
29 end
30 always @ (*) begin
31   case(control)
32     2'b00: tmp <= data;
33     2'b01: tmp <= data ^ key;
34     2'b10: tmp <= ~data;
35     2'b11: tmp <= data ^ {pattern[3], pattern[2:0] & counter
           };
36   endcase
37 end
38 always @ (posedge clk) begin
39   if (~reset) out <= 4'b0;
40   else out <= tmp;
41 end
42 endmodule
```

Line 27 is unreachable (and thus `pattern` will never be assigned $dc_0 - dc_3$) because the variable `counter` only takes on values 0-4. These X's are safe, and cannot be used to leak information, therefore are best left in the RTL to aid in logic optimization. A more interesting X-assignment occurs on Line 26, which can be re-written as:

$$\texttt{3'd4: pattern <= \{1, } dc_4\texttt{, } dc_5\texttt{, 1\};}$$

The assignment of $\{dc_4,\ dc_5\}$ to `pattern[2:1]` is reachable, however, by manual inspection, one can see that the only assignment influenced by `pattern` (Line 35) contains a bitwise AND between `counter` and `pattern[2:0]`, which prevents dc_5 from propagating further, but not dc_4! This is because when `counter = 3'd4`, Line 35 effectively becomes:

$$\texttt{2'b11: tmp <= data ^ \{1, } dc_4\texttt{, 0, 0\};}$$

In this example only 1 of 6 don't cares is dangerous and necessary to remove. In a design with hundreds of don't cares, it is expected that only a small subset is dangerous, which motivates why it is valuable for an X-analysis tool to take a fine-grain approach and distinguish between unreachable, reachable but non-propagating don't cares, and don't cares that have the potential to propagate to outputs or attacker observable points.

12.2.2 Automated Identification of Dangerous Don't Cares

Through the examples in the previous section, we have seen that a don't care bit, dc_i, is dangerous if an input sequence can be found for which circuit outputs differ depending on if dc_i is 0 or 1. For this to be possible, the statement assigning dc_i to a design variable must be reachable, and the value of the variable must propagate to circuit outputs. The problem of finding if such an input sequence exists has been formulated in [28] as a sequential equivalence checking problem. In [28], the analysis was performed to find X-bugs, not prevent Hardware Trojans, but like the Hardware Trojans we are proposing, X-bugs result from reachable X-assignments that affect primary outputs in the design.

One key difference between X-bugs and the proposed Trojan type is that in many designs, for example, a serial multiplier, or the Elliptic Curve Processor analyzed in Sect. 12.2.3, the values at the primary outputs of the unit during intermediate cycles in the computation typically don't matter, as long as the final computation result is correct. X's propagating to primary outputs during intermediate cycles generally aren't considered X-bugs if the final result is unaffected, however, information leakage can still occur during these intermediate cycles if the attacker can observe the primary outputs of the circuit.

The equivalence check is performed between two near identical versions of the design: one where $dc_i = 0$ and one where $dc_i = 1$. If the designs are identical under all possible input sequences, dc_i cannot possibly be used to leak design information.

We build upon this idea further by addressing the relationship between multiple don't cares in the design, and we formulate the problem in terms of combinational equivalence checking and state reachability analysis. For scalability reasons, our solution may over-approximate the set of don't cares classified as dangerous.

While combinational equivalence checking between two nearly identical designs is efficient and scalable, state reachability analysis is not. In the Elliptic Curve Processor case study presented in Sect. 12.2.3, we illustrate how commercial code-reachability tools can be used in place of symbolic state reachability analysis to re-classify don't cares erroneously marked as dangerous after combinational equivalence checking as safe.

Consider the generic example circuit in Fig. 12.2, where the sequential behavior has been removed by making all flip-flop inputs pseudo primary outputs (PPOs) and all flip-flop outputs pseudo primary inputs (PPIs). There are n don't care bits in the design, and it is clear that dc_i and dc_j have the ability to block each other

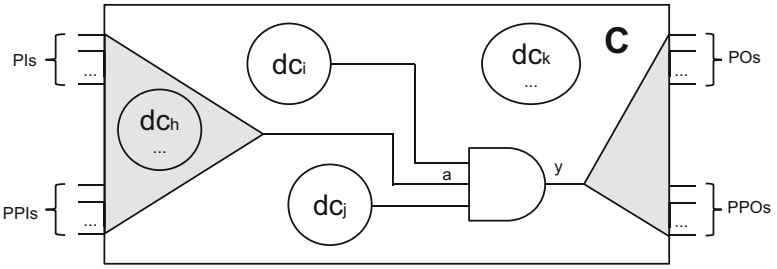


Fig. 12.2 Generic circuit with don't care bits

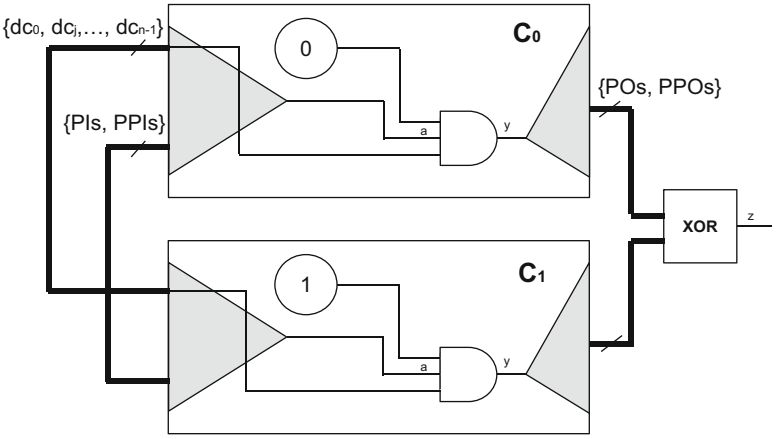


Fig. 12.3 Equivalence checking formulation

from propagating. dc_h is in the fan-in cone for signal a, and can also influence the propagation of dc_i and dc_j, while dc_k is completely independent from dc_i, dc_j, and dc_h.

Combinational equivalence checking can be performed between two versions of the original design: $C_{dc_i=0}$, and $C_{dc_i=1}$, by constructing the miter in Fig. 12.3 and checking the satisfiability of node z. If z is not satisfiable, then dc_i is safe. Otherwise, the equivalence checker returns a distinguishing input vector. Note that when analyzing dc_i, all remaining $n-1$ don't care bits are made primary inputs. This ensures the distinguishing input vector contains information about how the remaining don't care bits are constrained if dc_i is to successfully leak information.

Since we are not considering the sequential behavior of the design, the distinguishing input vector could require that the pseudo primary inputs be assigned a value that can never occur, in other words an *unreachable state*. State reachability analysis can be performed before analysis of all don't care bits, and a logic formula describing the set of unreachable states can be incorporated into the miter circuit to prevent the equivalence checker from finding distinguishing input vectors containing these states.

State reachability is a hard problem, but recent advances in model checking [29] and techniques such as [30], which over-approximate the set of reachable states, can aid in addressing non-trivial designs. Additionally, since don't cares can often be traced back to single-line assignments in the Verilog code, dead-code analysis and code reachability tools can help easily eliminate don't care assignments that are unreachable. For Trojan prevention, an over-approximation is ideal because it ensures that a dangerous don't care will never be classified as safe due to the elimination of a distinguishing input vector containing a state erroneously marked as unreachable.

12.2.3 Elliptic Curve Processor Case Study

We now present a case study in which manual inspection was used to identify don't cares in the control unit which provide an opportunity for a Trojan to leak all key bits. We then show how our automated prevention method classifies the don't cares which make this exploit possible as dangerous in addition to unearthing several previously unknown opportunities for information leakage.

Elliptic curve cryptography (ECC) is a public key cryptosystem whose fundamental operations use the mathematics of elliptic curves to perform key agreement and generate/verify digital signatures. ECC is currently used in SSH and TLS, and offers more security/key bit than RSA [31]. Like other cryptographic algorithms, ECC operations can be accelerated if implemented in hardware. Our case study examines a publicly available Elliptic Curve Processor (ECP) which performs the point multiplication operation optimized for an FPGA implementation [32].

Point multiplication is the fundamental operation on which all ECC protocols are built, and the reader should refer to [32] for more background on the mathematics behind this operation. Point multiplication takes as input, elliptic curve parameters, an initial point on the elliptic curve, P, and a secret k, and computes $G = [k]P$, which is P "added" to itself k times using the formulas for elliptic curve point addition and doubling. ECC is secure because it is very difficult to discover k knowing only G and P.

12.2.3.1 The Hardware Trojan

The Trojan inserted into the ECP allows an attacker who only is allowed to observe primary output signals to discover the secret k. This design contains a state machine with 38 states (shown in Fig. 12.4), multiple register files, and several custom arithmetic units used by various scheduled operations. The final point, G, is computed when State 38 is reached. The ECP Trojan exploits don't cares specified in a `case` statement during the assignment of control signals in the state machine logic.

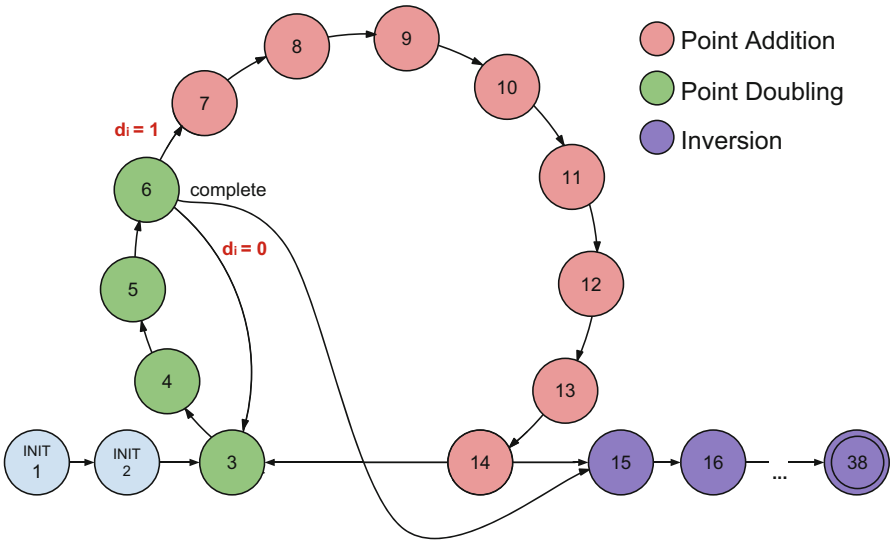


Fig. 12.4 ECP state machine

During State 15, don't cares assigned to control signals (`cwl` and `cwh`) propagate to a register bank address and write enable signal, effectively making the contents of the register bank unspecified during State 15. One of the registers is tied to a primary output signal, making it possible for an attacker to directly leak all the key bits to an observable output point. We refer the reader to [20] for details including code listings showing the exact design modifications required to leak the key bits.

Normally, an unknown value in a circuit output during an intermediate cycle in the computation is **not considered an error, because it does not affect the final point computed** during the point multiplication. We emphasize that with the knowledge of this new Trojan type, *any* X-propagation to primary outputs during *any* cycle must be prevented.

12.2.3.2 Automated X-Analysis

The ECP design has 572 primary input bits, 467 primary output bits, and 11,232 state elements, resulting in a gate count over 300,000. There are 538 don't care bits in the design analyzed by our tool. 282 correspond to assignments made during states 0–38 to bits in `cwl` and `cwh`, 33 correspond to the `default` assignments (*state* $>$ 38, which should be unreachable) of these signals, and 233 are from a `default` assignment in the `quadblk` module.

Table 12.1 Classification of don't cares

Row #	# don't care bits	Signal(s) affected
Class 1: definitely dangerous (35 bits)		
1	2	`cwh[4]`, `cwh[7]` , when `state`==15
2	1	`cwh[12]` , when `state`==2
3	32	`cwl`, for various states ≤ 38
Class 2: possibly dangerous (272 bits)		
4	6	`nextstate[5:0]`, when `state` > 38
5	23	`cwh[22:0]`, when `state` > 38
6	10	`cwh[9:0]`, when `state` > 38
7	233	`d[232:0]`, when `cwh[19:16]`==1 or `cwh[19:16]`==15
Class 3: definitely safe (231 bits)		

Combinational equivalence checking between two very similar designs scales well, and each don't care only requires a few minutes of analysis by ABC [33]. Using only combinational equivalence checking, the 538 don't cares are separated into two groups: definitely and possibly dangerous (307 bits), and definitely safe (231 bits).

Note that the dangerous don't cares in Row 1 of Table 12.1 correspond exactly to the don't cares selected by our original manual analysis to implement the Trojan! Rows 2 and 3 highlight additional don't cares which an attacker may be able utilize to leak up to 33 bits of information during various states. The distinction between definitely and possibly dangerous don't cares requires state reachability analysis, because the distinguishing input vector may contain an unreachable state. For example, the variable `nextstate` is assigned don't cares (see Row 4 of Table 12.1) only if the current state variable `state` is outside the 0–38 range, which a quick analysis of the RTL code will reveal can never occur.

Full blown state reachability analysis does not scale well, and we were unable to extract the exact set of unreachable states using ABC. However, we were able to determine that the lines of code containing the X-assignments in Rows 4–6 in Table 12.1 are unreachable using Spyglass, an RTL lint tool from Atrenta [34]. The exact Spyglass rules used to classify the don't cares in Rows 4–6 is given in [20].

We remove the opportunity for Trojan insertion by replacing the don't care bits listed in Table 12.1 with 0's and use Synopsys Design Compiler (ver I-2013.12-SP2) to synthesize the design and measure the area overhead of the modification. The don't cares in Rows 1–6 and Row 7 are from the `ecsmul` and `quadblk` modules, respectively.

Table 12.2 shows how replacing only dangerous, both dangerous and possibly dangerous, and all don't cares affects the area overhead of the `ecsmul` and `quadblk` modules. Even though using only combinational equivalence checking over-approximates the number of dangerous don't cares, being cautious and removing all don't cares in Classes 1 and 2 is still preferable to the 8 % area increase resulting from indiscriminately replacing every don't care bit (305 total) in `ecsmul`.

Table 12.2 Area overhead of specifying don't cares

	% area increase	
Don't cares defined	`ecsmul`	`quadblk`
Class 1	0.04	–
Classes 1 and 2	1.80	3.87
All don't care bits	8.00	3.87

12.3 Identifying Dangerous Unspecified Functionality

The don't care analysis technique proposed in the previous section relies on the maturity of combinational equivalence checking tools for RTL designs, making it hard to generalize to SystemC, C, and other high level modeling languages. Additionally, only unspecified functionality captured by don't care bits can be analyzed. This section builds upon the ideas in the previous section, but the proposed mutation-based method is more general since mutation testing is applicable to FSM, C, SystemC, TLM, RT, and gate-level models, only requiring that the model be executable and that a testing scheme exists.

The analysis methodology presented in this section randomly samples possible design modifications (known as mutations in mutation testing [35]). We filter out modifications that are not dangerous (do not affect unspecified or poorly tested functionality) by monitoring functional coverage and signals observable to the attacker/user. After our analysis, the verification team is presented with a list of design modifications ordered from most dangerous to least dangerous which are representative of functionality which either needs to be specified or better tested to ensure the absence of Trojans.

12.3.1 Background: Mutation Testing and Coverage Discounting

The goal of mutation testing is to gauge the effectiveness of the verification effort by inserting artificial errors (faults) into the design code then recording how many faulty versions of the design (mutants) are detected. Mutation analysis is motivated by the observation that if the test bench is unable to detect artificial errors, it is likely that real design errors are also going unnoticed.

Mutation testing has been used for software security analysis to verify security protocols, determine program susceptibility to buffer overflow attacks, and identify improper error handling [35, 36]. In the hardware domain, mutation testing is primarily used for test bench qualification [37]. Fault models and fault injection tools exist for SystemC [38], TLM [39], and RTL [40].

Two well-known drawbacks of mutation analysis are (1) long runtime and (2) large manual effort required to analyze undetected mutants, some of which may be redundant. Redundant mutants are those under all possible inputs, can never cause any change in the design "care" outputs.

Coverage Discounting [41] is a technique which identifies undetected mutants which cause changes in functional coverage. In doing so (1) redundant mutants are filtered out from analysis, (2) the remaining undetected mutants are associated with specific functional coverpoints making analysis easier, and (3) the coverage score is revised to reflect the error propagation and detection properties of the test bench.

Our technique builds upon Coverage Discounting by identifying mutants which cause changes in attacker-observable signals (in addition to those which cause changes in functional coverage) to filter out redundant mutants while highlighting mutants related to functionality vulnerable for use in information leakage Trojans.

To motivate how mutation testing can identify functionality susceptible for modification by information leakage Trojans, consider Listing 12.3 which gives the Verilog code describing the read behavior of the FIFO in Fig. 12.1. To illustrate the potential of mutation analysis to highlight the weakness in the verification infrastructure allowing this Trojan to exist undetected, consider a fault which changes the AND operator (highlighted in pink) to an OR operator in Line 2 of Listing 12.3. This fault causes `read_data` to update whenever the FIFO is not empty, even if a read operation is not occurring. If a read operation occurs, the read pointer will increment as seen in Line 7 of Listing 12.3 and in the following cycle, even if `read_enable == 0`, `read_data` will be updated with the value of the next FIFO item.

Listing 12.3 FIFO Read Behavior

```
1 //Memory Access Behavior
2 if (read_enable && !buffer_empty)
3   read_data <= mem[read_ptr];
4 ...
5 //Pointer Update Behavior
6 if (read_enable && !buffer_empty)
7   read_ptr <= read_ptr + 1;
```

During testing, it is likely that a read operation will occur, but the FIFO will not immediately become empty, meaning the spurious updating of `read_data` can be observed if the waveforms of the fault-free and faulty design are compared. However, it is unlikely that this fault will cause any tests to fail since the fault does not cause the read pointer to spuriously update, and when `read_enable == 0`, the test bench has no incentive to check the value of `read_data`. Notice that the functionality affected by this fault is useful for an attacker because:

1. Observable signals at the boundary of the main module deviate from the fault-free version during testing (indicating that information can be leaked during normal operation without requiring the attacker to force the design into a rare state)
2. The fault is undetected

The methodology presented in Sect. 12.3.2 would flag this fault for analysis, forcing the verification team to define behavior for the `read_data` signal when `read_enable == 0` then write a test case or checker for this behavior in order to detect the fault, resulting in an improved test bench able to detect the Trojan in Fig. 12.1.

12.3.2 Identification Procedure

In a security context, for a fault to be dangerous, it must be (1) undetected by the test suite and (2) cause changes in attacker-observable signals. Figure 12.5 shows how undetected faults can be classified based on their influence over attacker-observable signals and functional coverage. Dangerous faults fall into Regions A and B.2 in Fig. 12.5.

The labeling of attacker-observable signals depends on the design and attack model. For example, if an attacker can run a malicious user-level software program which interfaces with the hardware Trojan, certain registers will be marked as attacker-observable in addition to network interfaces. If the design being analyzed is a peripheral or co-processor, and it is assumed the main processor may contain a Trojan, the bus interfaces between modules are considered attacker-observable.

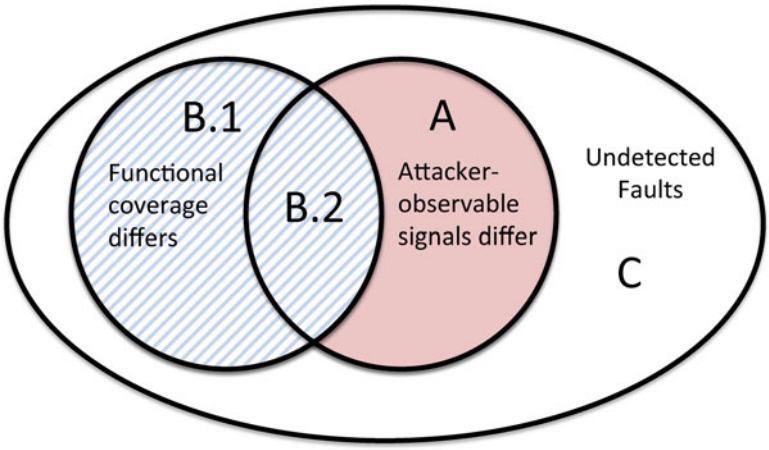


Fig. 12.5 Scenarios for undetected faults

If the attacker has physical access to the device, then all chip output pads are attacker-observable. A key point to note is that even if the correct values of some attacker-observable signals are unknown to the verification team, our technique only requires discovering differences in the simulation trace between the faulty and fault-free designs.

What about undetected faults affecting specified functionality (Regions B.1 and B.2 in Fig. 12.5)? The faults in Region B.1 do not affect attacker-observable signals but should be examined because they indicate design functionality is not adequately tested! The faults in Region B.1 cause the coverage score to be revised downward during Coverage Discounting [41]. Discounting separates faults affecting design functionality from redundant faults by recording changes in functional coverage caused by each fault. Discounting can be applied to any design where it is possible to define and record functional coverage.

We are able to add our analysis to the existing Coverage Discounting flow with only the additional overhead of tracking attacker-observable signals. The following flow both identifies test bench weakness affecting specified functionality and highlights dangerous unspecified functionality:

1. Record values of attacker-observable signals and functional coverage in the original design during all tests
2. Analyze the design and generate a set of faults, then inject each fault and re-run all tests, recording the same information as in Step 1
3. Only examine **undetected faults** (ones which do not cause any tests/assertions to fail)

The following details the actions that should be taken for every **undetected fault**, based on the region in Fig. 12.5 the fault belongs to:

- **Region A:** Functional coverage did **not** change under the fault, but a change in some attacker-observable signals occurred. It is likely that the fault affects **unspecified** design functionality susceptible to the insertion of information leakage Trojans. Behavior for the functionality affected by the fault must be specified, and then the test bench must be improved to check this newly defined behavior.
- **Region B (Regions B.1 and B.2):** Functional coverage changes under the fault meaning **specified** design functionality has been modified and this modification has gone unnoticed by the test bench. This indicates a weakness in the test bench, and the verification engineer must examine why this change in functionality went undetected and fix the test bench.
- **Region C:** Neither functional coverage nor attacker-observable signals change meaning the fault is likely redundant (for example, changing the loop condition in `for(x=0;x<10;x++)` to `for(x=0;x!=10;x++)`), and is not worth examining.

Although less than the total number of undetected faults, the number of undetected faults in Region A of Fig. 12.5 can still be too costly to completely analyze. By computing metrics for each fault such as the number of attacker-observable bits

differing, the total time attacker-observable signals differ, and the number of *distinct* tests producing differences in attacker-observable signals fault ranking metrics can prioritize the most dangerous faults for analysis first. We refer the reader to [21] for details on these ranking heuristics.

12.3.3 UART Communication Controller Case Study

We analyze a UART (universal asynchronous receiver/transmitter) design from OpenCores [42] using the methodology presented in Sect. 12.3.2. After analysis of just four of the most dangerous undetected faults returned by our method, we identify unspecified bus functionality and poorly tested interrupt functionality vulnerable to the insertion of information leakage Trojans. After further defining portions of the bus specification and correcting an error in the interrupt checker, the test infrastructure is able to detect these faults as well as an example Trojan inserted in the UART bus functionality.

The test bench used in this case study is a propriety OVM-based suite provided by an EDA tool vendor consisting of 80 directed tests with constrained random stimuli, functional checkers, and 846 functional cover points. This test bench is representative of a typical mature regression suite. There are 38 attacker-observable bits. 32 bits belong to the `wb_dat_o` signal, which is the data bus the UART places values onto when the bus master (often a processor core) issues read requests. The signals `wb_ack_o`, `int_o`, and `baud_o`, are single bit signals which acknowledge bus transactions, signal interrupts, and define the baud rate, respectively. These 35 signals comprise the interface to the processor core, while the remaining 3 attacker-observable signals are the off-chip serial output, request to send, and data terminal ready signals. For this experiment, we make the assumption that the attacker is able to see all 38 signals.

Mutation analysis is performed using the commercial mutation analysis tool Certitude [40]. Certitude faults are simple modifications made to the design source code, for example replacing an AND operator with an OR operator, or tying a module port to a static 0 or 1. 1183 faults are injected one by one in the design, and tests are run until the fault is detected. Out of the 1183 faults, 110 are not detected by any of the 80 tests. The classification of these faults into Regions A, B, and C in Fig. 12.5 is presented in Table 12.3. Using our methodology, the number of faults requiring manual analysis (those in Regions A and B) is reduced from 110 to 32.

Table 12.3 Categorization of undetected faults

110 undetected faults		
Region A	Region B	Region C
30 faults	2 faults	78 faults

12.3.3.1 The Wishbone Bus Trojan

The three most dangerous faults determining by our ranking heuristics relate to Wishbone Bus [43] interface signals. All faults affect the assignment of the output enable (`oe`) signal, causing spurious changes on the data output bus. In the original design, `oe` is only set during a valid read transaction. Under the three faults, `oe` is incorrectly set during write transactions, when the UART slave is not selected, and when a valid bus cycle isn't in progress. These faults are undetected because during these cycles the bus master never captures data from `wb_dat_o`, so the extra data bus changes never cause incorrect data to be read from or written to the UART registers.

This analysis indicates the UART design may be infested with a Trojan that can leak information with impunity on the data bus as long as a valid read transaction is not taking place and the test bench would be none the wiser! Specifically, the Trojan can take advantage of the fact that the value of the output data on the bus (`wb_dat_o`) is **unspecified** during a write transaction or invalid cycle.

We implement one possible bus Trojan by changing the assignment of `oe` to one of the undetected faulty versions which allow `wb_dat_o` to spuriously update, then leak the value `0xdeadbeef` over the bus only during write transactions and invalid cycles.

Figure 12.6 illustrates the ability of the Trojan to leak 32 bits of data during every write transaction while not interfering with read transactions. For example, at 135 ns, the UART responds to a read request with the correct data, not `0xdeadbeef`. Simply placing `0xdeadbeef` on the data bus is good for illustrative purposes, but may not be useful to an attacker. One should note that *any* 32-bit value can be assigned to `wb_dat_o`, including other secret internal design signals.

This Trojan is fundamentally different from Trojans relying on rare triggering conditions for stealth as it is active during every write transaction, which is certainly not a rare design state, as evidenced in Fig. 12.6. It is unlikely that this Trojan would be detected by existing methods targeting the identification of rarely used logic.

To detect these dangerous faults, the following additional check is added to the existing bus protocol checker: `wb_dat_o` cannot change unless the design has been reset, or a read request is being acknowledged. In addition to detecting the three faults, the Output Enable Bus Trojan is also detected.

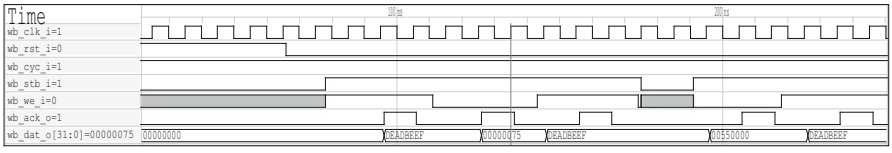


Fig. 12.6 Output enable Trojan waveform for bus protocol test

In a traditional verification setting, it would be unnecessary and cumbersome to add this additional check, and the three faults would be considered a waste of time to analyze because they do not affect the correctness of normal read/write operations. We would like to highlight the relationship between undetected faults affecting attacker-observable signals and hardware Trojans, providing motivation to analyze and improve the test bench to detect these seemingly meaningless artificial errors.

Through mutation analysis, which is a random sampling of very specific design modifications, we have actually found a more general class of Trojan, the bus protocol Trojan. The bus protocol Trojan takes advantage of unspecified functionality such as data bus values when no valid transactions are taking place, and the value of the data output bus during a WRITE cycle. The FIFO Trojan actually belongs to this class of Trojan.

12.3.3.2 Interrupt Output Signal Checker Bug

After improving the test bench to detect the three faults related to the Wishbone bus, the next most dangerous fault affects *specified* functionality, and belongs to Region B.2 in Fig. 12.5. Interestingly, this fault is not highlighted by Coverage Discounting, but is by our technique.

The UART uses a single bit signal, `int_o`, to notify the host processor of pending interrupts. There are five different events which can cause an interrupt, and the Interrupt Identification Register (IIR) indicates the highest priority interrupt currently pending. A commonly used interrupt is the received data available (RDA) interrupt, which fires when a threshold number of characters is received.

The fault causes `int_o` to become unknown for many cycles during 49 of the 80 tests. More specifically, the fault causes the RDA interrupt pending signal `rda_int_pnd` to become X instead of 1 under certain conditions, making it possible to selectively suppress the RDA interrupt (and consequently `int_o`) without the test bench noticing.

Although the test bench checks if IIR bits are set correctly when conditions for each interrupt type are met, and *most of the time* checks that `int_o` reflects the IIR interrupt pending bit within ten clock cycles, the behavior of `int_o` is not checked if `int_o` becomes X. Moreover, even if a Trojan set `int_o` to a non-X value in order to leak information, as long as `int_o` becomes both 0 and 1 within 10 clock cycles, the interrupt checkers would not notice that `int_o` is changing spuriously with respect to the IIR interrupt pending bit. This oversight in the test bench is an example of poorly tested *specified* functionality, since the value of `int_o` is clearly being checked in the interrupt checker, but not thoroughly enough.

It is interesting to note that this fault did not cause a change in functional coverage, perhaps suggesting that the coverage model is not detailed enough to highlight meaningful verification holes in the interrupt functionality illustrating the potential of our analysis technique to highlight and qualify the verification of important design functionality outside of the coverage model.

12.4 Trojans in Partially Specified On-chip Bus Functionality

The general method to identify dangerous unspecified functionality in any type of design requires mutant simulation and analysis, which is expensive but necessary if one cannot identify dangerous unspecified functionality directly by inspection. Since bus systems are characterized by well-defined protocols and set of common topologies, we are able to directly present a general model for dangerous unspecified bus functionality in this section. This work takes inspiration from the Wishbone bus Trojan presented in the previous section, and generalizes the Trojan to other bus protocols and more complex bus topologies.

The ability to manipulate the bus system is extremely valuable to an attacker since the bus controls communication between critical system components. A denial of service Trojan halting all bus traffic can render an entire SoC useless. Any information transferred to/from main memory, the keyboard, system display, network controller, etc. can be passively captured or actively modified by Trojans inserted in the interconnect.

There exist many different bus protocols designed to optimize different design parameters such as area/timing overhead, power consumption, and performance [44]. Regardless, all protocols employ signals to mark when valid bus transactions occur and handshakes to provide rate-limiting capabilities, meaning valid and idle bus cycles can be clearly differentiated. While bus protocols clearly define the desired values for each data or control signal during *valid* transactions, the values of these signals during idle cycles are **unspecified** and largely ignored by bus protocol checkers, formal verification properties, and scrutiny during simulation-based verification. Trojan behavior during these cycles will not be detected by traditional verification methodologies. For example, Fig. 12.7 shows the VALID/READY handshake used by each channel in the widely used AXI4 protocol. When VALID is LOW, the information lines can take on *any* value, including Trojan information.

The bus Trojans we propose in this section operate entirely within idle bus cycles, with the goal being to provide a covert communication channel built upon existing

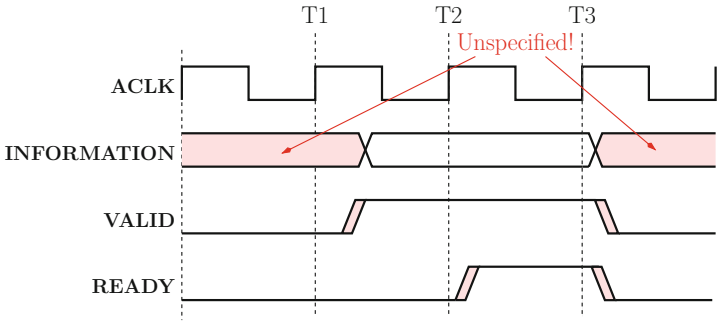


Fig. 12.7 AXI bus protocol VALID/READY handshake [45]

bus infrastructure. This Trojan channel can be used to connect Trojan components spread across the SoC in addition to enabling information leakage from legitimate components not possible in the original design. Unlike previously proposed bus Trojans, which lock the system bus, modify bus data, and allow unauthorized bus transactions [5, 46], our Trojans never hinder normal bus functionality or affect valid bus transactions.

12.4.1 Threat Model

Since a covert communication channel is useless without a sender and receiver of information, we assume that at least one component connected to the system bus contains a Trojan utilizing the information received on the channel, and that there is another Trojan to either leak data from the component it resides in or snoop bus data otherwise not visible to the receiver and send it over the channel.

Since the proposed Trojans operate entirely within unspecified bus functionality they cannot suppress or alter valid bus transactions, meaning any Trojan actions must be contained to cycles and signals where no valid transactions are occurring. Although it is possible for the Trojan to create new bus transactions adhering to the bus protocol during unused cycles, verification infrastructure often includes bus checkers which count and log all valid bus transactions. For this reason, our proposed Trojans do not suppress, alter, or create valid bus transactions, but instead re-use existing bus protocol signals to define a new "Trojan" bus protocol allowing communication between different malicious components across the SoC.

It is assumed the Trojans are inserted in the RTL code or higher-level model, meaning no golden RTL model exists to aid in Trojan detection at later stages in the design cycle. A complex SoC requires hundreds of engineers to design and test, and relies on third party IP cores and tools to meet time to market demands. A single rouge design engineer or malicious 3rd party IP or CAD tool vendor has the potential to implement a Trojan communication channel.

12.4.2 Trojan Communication Channel

The structure and size of the Trojan channel circuitry depends on the following:

1. **Bus Topology:** Determines necessity of FIFO and extra Leakage Conditions Logic at receiver interface
2. **Bus Protocol:** Defines Leakage Conditions Logic and selection of signal(s) to mark valid Trojan transactions
3. **Trojan Channel Connectivity:** Channel can be one-way or bi-directional, contain an active or snooping sender, and involve information leakage between two masters, two slaves, or a master and a slave

4. **Data Width of Trojan Channel** (k)**:** number of bits leaked during a Trojan transaction
5. **FIFO Depth** (d)**:** FIFO used to buffer Trojan channel data if the receiver is busy accepting valid bus transactions

Bus topology and protocol are selected by the system designer, whereas Trojan channel connectivity is chosen by the attacker. Data width (k) and Trojan FIFO depth (d) are parameters selected by the attacker to trade-off performance and overhead of the Trojan channel. The black-colored components in Fig. 12.8 are necessary to implement a Trojan communication channel for a shared bus topology, which is shown in Fig. 12.9a. For this case, the Data and Control lines from the sender component are directly visible at the receiver. The red-colored components in Fig. 12.8 show the extra circuitry required to implement the channel in an interconnect with a MUX based topology, which is shown in Fig. 12.9b.

The sender and the receiver can be any master or slave component on the interconnect. The goal of the Trojan channel is to use *only pre-existing* interconnect

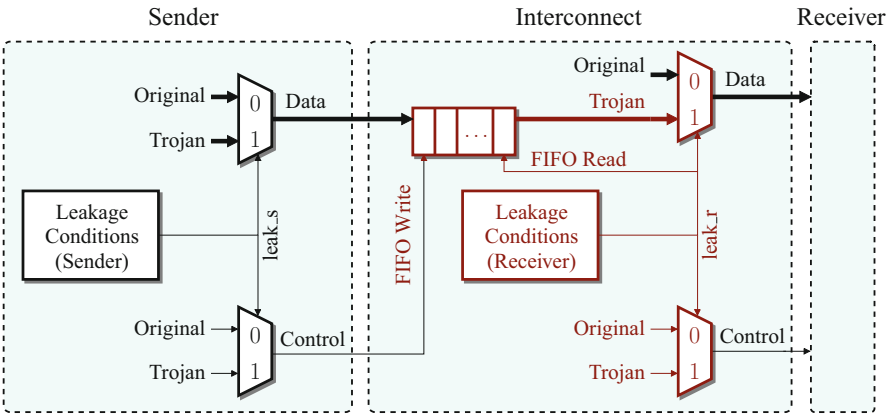


Fig. 12.8 Trojan channel circuitry

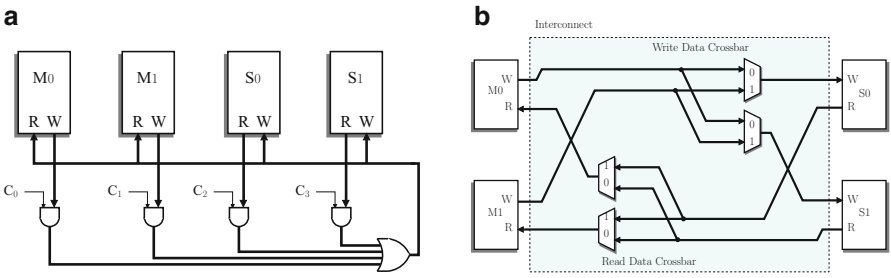


Fig. 12.9 Bus topologies on opposite ends of the area v. throughput spectrum. (**a**) Shared R/W data channels [44]. (**b**) Concurrent data channels [47]

interfaces to pass data from the sender to the receiver. For example, the line labeled Data in Fig. 12.8 on the sender's side could be the write data or read/write address port if the sender is a bus master and the read data port if the sender is a bus slave and vice versa for the Data on the receiver's side.

Since the Trojan data is transmitted using the same lines as normal bus traffic, additional signaling must mark when valid Trojan data is being transmitted. These signals are labeled as Control in Fig. 12.8, and like the Trojan data, are mapped to pre-existing data/address/control signals, meaning no additional interface ports are created. The Leakage Conditions Logic is protocol dependent and examines signals at the sender's interconnect interface to determine when it is "safe" to replace the original bus signal values with Trojan values.

12.4.2.1 Topology Dependent Trojan Channel Properties

All bus signals can be classified as address, data, or control signals, and additionally classified as belonging to read and/or write functionality. The interconnect topology specifies the degree of parallelism between the different categories of bus signals, and the connectivity between masters and slaves [44].

Figure 12.9a and b show the read and write data channels for topologies sitting at opposite ends of the area efficiency and channel throughput trade-off. Figure 12.9a is the most area efficient, but can only support a single transaction at a time, whereas Fig. 12.9b contains significantly more circuitry, but can support multiple simultaneous transactions.

In Fig. 12.9a, all read and write transactions are visible to all bus components, meaning no Trojan circuitry is required to simply snoop bus data. If a Trojan bus component wishes to send information, the black-colored circuitry inside the sender block of Fig. 12.8 is required. In Fig. 12.9b, data is not visible to a component uninvolved in the transaction. Unlike Fig. 12.9a, forming a channel between two slaves or two masters requires extra circuitry inside the interconnect, shown in red in Fig. 12.8. Because the signals at the sender's interconnect interface are not visible at the receiver's interface and vice versa, new leakage conditions are required, which monitor the receiver's interface and determine when it is safe to leak data without altering valid bus transactions. Signals available at the receiver's interface must also be selected to implement the Data and Control lines. The FIFO is necessary because leakage conditions at the sender and receiver may not occur simultaneously.

12.4.2.2 Protocol Dependent Trojan Channel Properties

The specifics of the Leakage Conditions Logic, which produces *leak_s* and *leak_r*, and the selection of Data and Control signals depend on the bus protocol used. Because of the similarities between various bus protocols, a general procedure for determining the Leakage Conditions Logic and the selection of Data and Control signals can be given.

Data Signal Selection In order to remain stealthy, the Trojan cannot create additional signals to transmit data, and must send data via pre-existing signals in the bus protocol. Being that the primary purpose of a bus is to transmit data, all bus protocol/topology combinations have signals that are suitable for sending/receiving Trojan data. In a protocol with separate read and write data signals, selection depends on if the Trojan Sender/Receiver resides in a master or slave component, since masters drive write data and observe read data signals, and vice versa for slave components. If the Trojan Sender resides in a master component, the read and write address signals can also be used to send Trojan data.

Leakage Conditions Logic Since pre-existing bus signals are used to transmit Trojan data, logic ensuring that normal bus operation is not compromised by the Trojan is necessary. The Leakage Conditions Logic examines protocol control signals to identify when Trojan Data signals are not being used to transmit *valid* data, and have unspecified values. Every bus protocol clearly defines the conditions for which data, address, and error reporting signals are valid. Some protocols, such as AXI4, designate a "valid" signal for each data channel, while others such as APB use the current state within the protocol to identify which signals are valid. *leak_s* is set when the Trojan Sender has data to transmit and the Data signals are not involved in a valid transaction. If the Trojan Sender is leaking valid bus transactions instead of actively sending information, then *leak_s* is not needed. *leak_r* is set when there are items in the Trojan FIFO and the Data signals at the receiver interface are not currently involved in a valid transaction.

Control Signal Selection When a Trojan Data signal is not being used in a valid bus transaction, its value is unspecified. During idle bus cycles, either Trojan data is being transmitted, or the bus is truly idle, and no data (Trojan or valid) is sent. To distinguish between these two cases, existing bus signals are selected to be Trojan Control signals, which mark when Trojan data is on the bus. The criteria for selecting these signals and their corresponding values is that when *leak_s/leak_r* is asserted, the normal behavior of the signal is predictable, but also unspecified. For most protocols, control signals are good candidates because they often are unused during idle cycles, yet their values remain static when idle for a given implementation.

We refer the reader to [22] for a detailed comprehensive presentation of the selection of Trojan Data and Control signals and Leakage Conditions Logic for the AMBA AXI4 protocol, and instead explain the reasoning behind a single selection of these signals for the AXI4-Lite based example system in the following section.

12.4.3 AXI4-Lite Interconnect Trojan Example

The system shown in Fig. 12.10 is created to verify the AXI4-Lite Interconnect Fabric through RTL simulation. The two slaves are simple 8-bit adder coprocessors which receive three operands to add via the interconnect from three processors.

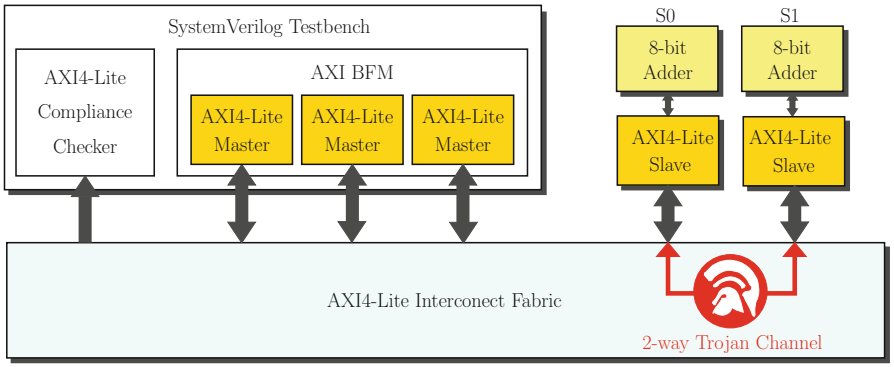


Fig. 12.10 AXI4-lite example system verification infrastructure

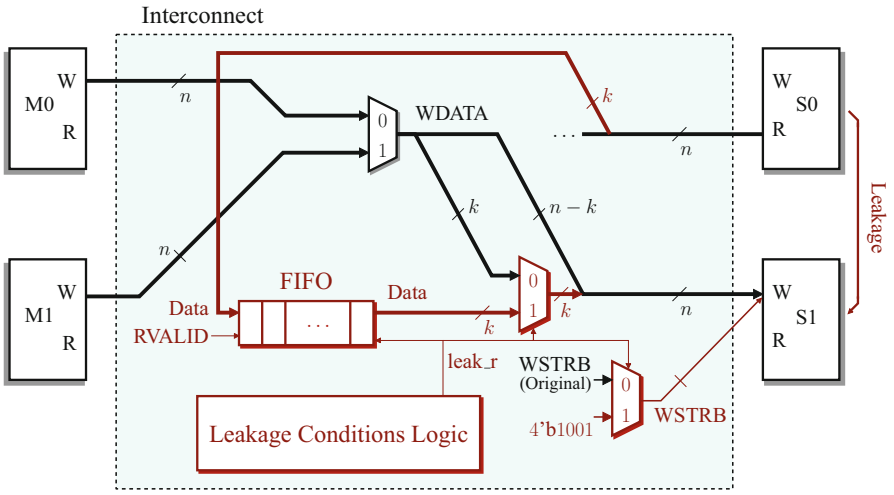


Fig. 12.11 Trojan channel logic for AXI4-lite interconnect

Since the specifics of the main processors are irrelevant, in the example infrastructure, they are replaced by AXI4-Lite bus functional models (BFMs) from [48]. Additionally, AXI4-Lite assertions packaged by ARM for protocol compliance checking [49] are active during system simulation.

The AXI4-Lite Interconnect Fabric IP block used is the LogiCORE IP AXI Interconnect (v1.02.a) from Xilinx [47] configured in Shared-Address Multiple-Data (SAMD) mode (the topology shown in Fig. 12.9b).

The AXI4-Lite Interconnect IP in Fig. 12.10 is infected with two copies of the circuitry shown in red in Fig. 12.11 to allow S1 to snoop on read requests for S0 and vice versa. Without the Trojan, the read data channel for S0 is not visible to S1 and vice versa. We now provide the reasoning behind the selection of Data and Control signals and Leakage Conditions Logic for the Trojan channel circuitry shown in Fig. 12.11.

Data Signal Selection k-bits of the n-bit AXI4-Lite write data signal (WDATA) are chosen to transfer the leaked information stored in the FIFO to the bus interface at S1 because S1 is a slave component, driving data signals on the read data channel, and receiving signals on the write data channel. There is no Trojan Data signal for the Sender since k bits of valid read data from S0 are being snooped, not actively transmitted.

Leakage Conditions Logic In AXI4 and AXI4-Lite 5 independent transaction channels are seen at the interface of every master and slave: the read address channel, read data channel, write address channel, write data channel, and write response channel [45]. Each channel uses a VALID/READY handshake signal pair, as shown in Fig. 12.7, to indicate when the receiver is ready to process bus data, and to mark when valid data is on the bus. Since the Trojan Data is being transmitted using the write data signal, the write data channel valid signal (WVALID) can be used to define *leak_r* as follows: $leak\_r = troj\_data\_ready$ & !WVALID.

Control Signal Selection The WSTRB signal is used to indicate when leaked data is on the bus. Normally WSTRB $==$ 1, because the bus data width is 32-bits, however the adder coprocessors have registers which are 8-bits wide. Because of this, any values for WSTRB with Hamming weight greater than 1 have no functional relevance in this system and are unused, making such values good candidates for marking when leaked data is on the bus. In this example Trojan channel, when information is leaked on WDATA, WSTRB $==$ 9.

The waveform in Fig. 12.12 first demonstrates how 3 read data responses (values 42, 15, then 14) from S1 are snooped and routed to S0's write channel, then shows a single read data response (value 96) from S0 routed to S1's write channel, and finally another read data response from S1 (value 13) leaked to S0. All Trojan transactions are highlighted in red in Fig. 12.12.

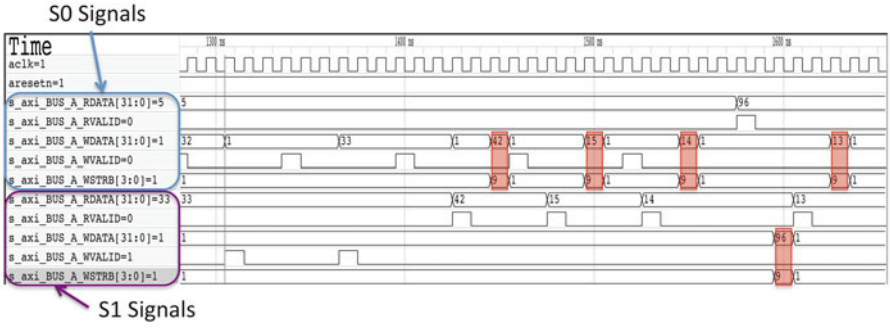


Fig. 12.12 2-Way information leakage waveform

Table 12.4 Trojan-free design results (after place and route)

Configuration	# FF	# LUT	# BRAM	Frequency [MHz]
3 masters 2 slaves	1814	2474	2	250
4 masters 6 slaves	3071	4247	3	250

Table 12.5 Area overhead of 2-way HW-Trojan channel

Data width	FIFO depth	% increase in FF		% increase in LUT	
		3M2S	4M6S	3M2S	4M6S
2	2	0.8	0.5	0.9	0.4
	4	1.1	0.7	1.5	0.6
	8	1.4	0.8	1.8	1.1
4	2	1.0	0.6	1.4	0.7
	4	1.3	0.8	2.0	0.8
	8	1.7	1.0	2.0	1.5
8	2	1.4	0.8	1.8	1.0
	4	1.8	1.0	2.4	1.2
	8	2.1	1.2	3.0	1.7

For AXI4-Lite, there are over 50 assertions monitoring bus signals during simulation, and none of them are violated even when information is flowing through the Trojan channel.

12.4.3.1 Overhead

To determine the area and timing overhead of implementing a 2-way Trojan channel between S0 and S1, the SystemVerilog Testbench in Fig. 12.10 is replaced by several simple bus masters. Table 12.4 shows results for the Trojan-free design, after placement and route, assuming 3 masters and 2 slaves (labeled as 3M2S) as well as 4 masters and 6 slaves (labeled as 4M6S) for a Virtex-7 FPGA (7vx330t-3).

Table 12.5 illustrates how the selection of Trojan channel parameters Data Width (k) and FIFO Depth (d) affects the results. The Trojan channel does not affect the operating frequency of the design, and stays within 3 % of the original FF and LUT utilization. As the number of masters and slaves increases, the interconnect and overall design area increases, but the size of the Trojan circuitry does not change.

The Trojan channel is easier to hide as the complexity of the interconnect and the number of components connected increases. The master and slave components used to generate the results in Tables 12.4 and 12.5 are far simpler than those in a typical SoC, so the results in Table 12.5 give a loose upper bound on the expected percentage of area increase caused by the Trojan channel in a modern design.

12.5 Conclusion

In this chapter we have addressed the threat of Hardware Trojans in unspecified design functionality. Due to the complexity of modern chips, a design specification usually only defines a small fraction of behavior. Traditional verification techniques only focus on ensuring the correctness of specified behavior, meaning any modifications or bugs (malicious or accidental) only affecting unspecified functionality will likely go undetected.

We have shown how this verification hole allows an attacker with the ability to modify the design to stealthily undermine the security of a system. We have shown that all secret key bits in an Elliptic Curve Processor can be leaked by only modifying RTL don't cares, and that it is possible to create a covert Trojan communication channel on top of existing on-chip bus infrastructure for common bus protocols by only modifying the on-chip bus interface signals when the channel is idle.

By viewing security as an extension of the verification problem we develop several analysis methodologies based on existing techniques such as equivalence checking and mutation testing which both prevent Trojans and increase confidence in the correctness of specified design functionality. These techniques include a Trojan prevention methodology based on equivalence checking that classifies all don't care bits in a design as dangerous or safe and a mutation testing based methodology capable of identifying dangerous unspecified functionality regardless of the abstraction level or class of design analyzed. Because unspecified functionality is by nature unknown, there is still much work to be done in fully exploring the scope of the Trojan threat in this space.

References

1. M. Dale, Verification crisis: managing complexity in SoC designs. EE Times (2001) [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1215507
2. S. Adee, The hunt for the kill switch. IEEE Spectr. **45**(5), 34–39 (2008)
3. S. Mitra, H.-S.P. Wong, S. Wong, The Trojan-proof chip. IEEE Spectr. **52**(2), 46–51 (2015)
4. Y. Shiyanovskii, F. Wolff, A. Rajendran, C. Papachristou, D. Weyer, W. Clay, Process reliability based trojans through NBTI and HCI effects, in *2010 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)* (IEEE, Anaheim, 2010), pp. 215–222
5. L.-W. Kim, J.D. Villasenor, Ç.K. Koç, A Trojan-resistant system-on-chip bus architecture, in *Proceedings of the 28th IEEE Conference on Military Communications*. Ser. MILCOM'09 (2009), pp. 2452–2457
6. S.T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, Y. Zhou, Designing and implementing malicious hardware, in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (USENIX Association, Berkeley, CA, 2008), pp. 5:1–5:8
7. L. Lin, W. Burleson, C. Paar, Moles: malicious off-chip leakage enabled by side-channels, in *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, November (2009), pp. 117–122

8. G.T. Becker, F. Regazzoni, C. Paar, W.P. Burleson, Stealthy dopant-level hardware trojans, in *Cryptographic Hardware and Embedded Systems (CHES)*. Ser. Lecture Notes in Computer Science, vol. 8086, ed. by G. Bertoni, J.-S. Coron (Springer, Berlin, Heidelberg, 2013), pp. 197–214
9. M. Tehranipoor, F. Koushanfar, A survey of hardware trojan taxonomy and detection. IEEE Des. Test Comput. **27**(1), 10–25 (2010)
10. R.S. Chakraborty, S. Narasimhan, S. Bhunia, Hardware trojan: threats and emerging solutions, in *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International* (IEEE, San Francisco, 2009), pp. 166–171
11. C. Krieg, A. Dabrowski, H. Hobel, K. Krombholz, E. Weippl, Hardware malware. Synth. Lect. Inf. Secur. Priv. Trust **4**(2), 1–115 (2013)
12. A. Waksman, S. Sethumadhavan, Silencing hardware backdoors, in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, Ser. SP'11 (2011), pp. 49–63
13. S.S. Ali, R.S. Chakraborty, D. Mukhopadhyay, S. Bhunia, Multi-level attacks: an emerging security concern for cryptographic hardware, in *2011 Design, Automation Test in Europe* (2011), pp. 1–4
14. S. Bhasin, J.L. Danger, S. Guilley, X.T. Ngo, L. Sauvage, Hardware Trojan horses in cryptographic IP cores, in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, August (2013), pp. 15–29
15. D. Agrawal, et al., Trojan detection using IC fingerprinting, in *IEEE Symposium on Security and Privacy*, 2007
16. A. Waksman, M. Suozzo, S. Sethumadhavan, FANCI: Identification of stealthy malicious logic using Boolean functional analysis, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS'13* (ACM, New York, 2013), pp. 697–708
17. D. Sullivan, J. Biggers, G. Zhu, S. Zhang, Y. Jin, FIGHT-metric: functional identification of gate-level hardware trustworthiness, in *Proceedings of the 51st Annual Design Automation Conference, DAC'14* (ACM, New York, 2014), pp. 173:1–173:4
18. J. Zhang, F. Yuan, L. Wei, Z. Sun, Q. Xu, VeriTrust: verification for hardware trust, in *Proceedings of the 50th Annual Design Automation Conference, DAC'13* (ACM, 2013), pp. 61:1–61:8
19. M. Hicks, et al., Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP'10* (IEEE Computer Society, Washington, 2010), pp. 159–172
20. N. Fern, S. Kulkarni, K.-T. Cheng, Hardware Trojans hidden in RTL don't cares - Automated insertion and prevention methodologies, in *Test Conference (ITC), IEEE International*, October (2015), pp. 1–8
21. N. Fern, K.-T. Cheng, Detecting hardware trojans in unspecified functionality using mutation testing, in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD'15* (IEEE Press, Piscataway, 2015), pp. 560–566
22. N. Fern, I. San, Ç.K. Koç, K.T. Cheng, Hardware Trojans in incompletely specified on-chip bus systems, in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March (2016), pp. 527–530
23. R.A. Bergamaschi, D. Brand, L. Stok, M. Berkelaar, S. Prakash, Efficient use of large don't cares in high-level and logic synthesis, in *1995 IEEE/ACM International Conference on Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers* , November (1995), pp. 272–278
24. M. Turpin, The dangers of living with an x (bugs hidden in your verilog), in *Boston Synopsys Users Group (SNUG)*, October 2003
25. L. Piper, V. Vimjam, X-propagation woes: masking bugs at RTL and unnecessary debug at the netlist, in *Design and Verification Conference and Exhibition (DVCon)*, 2012
26. H.Z. Chou, H. Yu, K.H. Chang, D. Dobbyn, S.Y. Kuo, Finding reset nondeterminism in rtl designs - scalable x-analysis methodology and case study, in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March (2010), pp. 1494–1499

27. Cadence conformal equivalence checker [Online]. Available: http://www.cadence.com/products/ld/equivalence_checker
28. M. Turpin, Solving verilog x-issues by sequentially comparing a design with itself. you'll never trust unix diff again! in *Boston Synopsys Users Group (SNUG)*, 2005
29. A.R. Bradley, SAT-based model checking without unrolling, in *Verification, Model Checking, and Abstract Interpretation* (Springer, Berlin/Heidelberg 2011), pp. 70–87
30. G. Cabodi, S. Nocco, S. Quer, Improving SAT-based bounded model checking by means of BDD-based approximate traversals, in *Design, Automation and Test in Europe Conference and Exhibition, 2003* (2003), pp. 898–903
31. J.W. Bos, J.A. Halderman, N. Heninger, J. Moore, M. Naehrig, E. Wustrow, Elliptic curve cryptography in practice, in *Financial Cryptography and Data Security* (Springer, 2014), pp. 157–175
32. C. Rebeiro and D. Mukhopadhyay, High performance elliptic curve crypto-processor for FPGA platforms, in *12th IEEE VLSI Design and Test Symposium*, 2008
33. ABC [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/
34. Atrenta spyglass lint tool [Online]. Available: http://www.atrenta.com/pg/2/
35. Y. Jia and M. Harman, An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
36. B. Breech, M. Tegtmeyer, L. Pollock, An attack simulator for systematically testing program-based security mechanisms, in *2006 17th International Symposium on Software Reliability Engineering*, November (2006), pp. 136–145
37. N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, F. Letombe, Functional qualification of tlm verification, in *2009 Design, Automation Test in Europe Conference Exhibition*, April (2009), pp. 190–195
38. P. Lisherness, K.T. Cheng, Scemit: a systemc error and mutation injection tool, in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, June (2010), pp. 228–233
39. N. Bombieri, F. Fummi, G. Pravadelli, A mutation model for the systemC TLM 2.0 communication interfaces, in *2008 Design, Automation and Test in Europe*, March (2008), pp. 396–401
40. Synopsys certitude [Online]. Available: https://www.synopsys.com/TOOLS/VERIFICATION/FUNCTIONALVERIFICATION/Pages/certitude-ds.aspx
41. P. Lisherness, N. Lesperance, K.T. Cheng, Mutation analysis with coverage discounting, in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March (2013), pp. 31–34
42. UART 16550 core [Online]. Available: http://opencores.org/project,uart16550
43. Wishbone bus [Online]. Available: http://opencores.org/opencores,wishbone
44. S. Pasricha, N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect* (Morgan Kaufmann Publishers Inc., Burlington, 2008)
45. *AMBA AXI and ACE Protocol Specification, Issue E*, ARM, 2013
46. L.-W. Kim, J.D. Villasenor, A system-on-chip bus architecture for thwarting integrated circuit Trojan horses, in *IEEE Transactions on VLSI Systems* **19**(10), 1921–1926 (2011)
47. *DS768: LogiCORE IP AXI Interconnect (v1.02.a)*, Xilinx Inc., March 2011
48. Axi4 bfm [Online]. Available: https://github.com/sjaeckel/axi-bfm
49. Amba 4 axi4, axi4-lite and axi4-stream protocol assertions bp063 release note (r0p1-00rel0), ARM [Online]. Available: https://silver.arm.com/browse/BP063

Chapter 13
Verifying Security Properties in Modern SoCs Using Instruction-Level Abstractions

Pramod Subramanyan and Sharad Malik

13.1 Introduction

In the era of Dennard-scaling, advances in computing performances were driven by transistor scaling. With each succeeding generation, transistors were smaller, faster, and more power-efficient than before [7, 17]. These gains were used to increase integrated circuit (IC) performance by designing more complex chips with a higher frequency of operation. The end of Dennard-scaling led to the multicore era where increases in performance were driven by parallelism rather than increased operating frequency [31]. However power and thermal constraints still limit performance and we are now in the dark silicon era where significant parts of an IC must be powered-off in order to stay within its power and thermal budgets [18].

Despite the technological limitations imposed by the dark silicon era, the demand for increased application performance and power-efficiency has not subsided and this has led to the rise of *accelerator-rich* system-on-chip (SoC) architectures [12]. Application-specific functionality is implemented using fixed-function or semi-programmable accelerators to obtain increased performance and power-efficiency. These accelerators are controlled and orchestrated by *firmware* which executes on programmable cores. Overall system functionality is implemented by this combination of firmware, programmable cores, and hardware components.

Figure 13.1 shows the structure of a modern SoC. It contains multiple programmable cores, accelerators, memories, and I/O devices connected by an on-chip interconnect [41, 44]. These components are organized into hierarchical modules, also referred to as intellectual property (IP) blocks. Typically each IP block contains a processor core, private memory, and several accelerators and I/O devices. Simpler IPs may contain only accelerators and/or I/O devices.

P. Subramanyan (✉) • S. Malik
Department of Electrical Engineering, Princeton University, Princeton, NJ, USA
e-mail: psubrama@princeton.edu; sharad@princeton.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_13

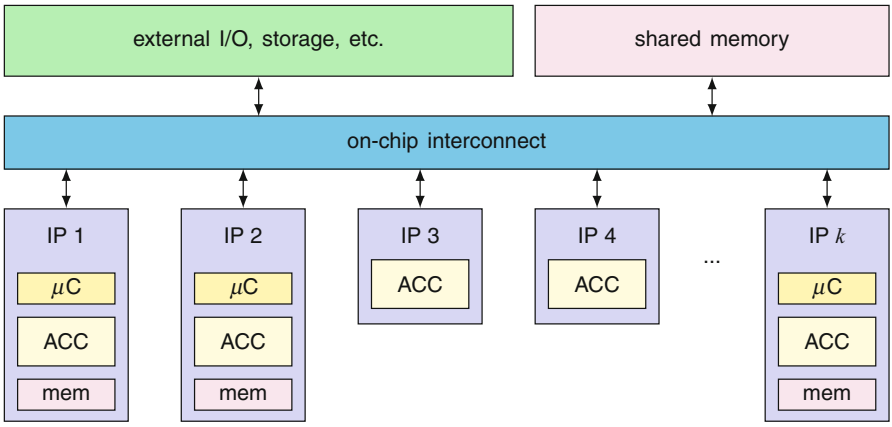


Fig. 13.1 Overview of a system-on-chip design

Due to time-to-market pressures imposed by the competitive environment, IP blocks are often obtained from semi-trusted or untrusted third-party vendors. This has two effects on the security verification problem: (1) the functionality of these blocks may not be fully or clearly specified (e.g., verification engineers may only have access to gate-level netlists instead of RTL designs) making verification more challenging, and (2) due to their untrusted nature, the designs may have subtle bugs which violate system-level security requirements of the SoC.

It is important to emphasize the role of firmware in modern SoC designs. Firmware is code that executes on programmable cores. It is shipped with the hardware and sits "below" the operating system and interacts closely with hardware accelerators, the on-chip interconnect, and I/O devices. System-level functionality including security features and protocols are *orchestrated* by firmware—firmware initiates operation of the hardware accelerators involved, collects and coordinates their responses, and manages resource acquisition and release.

13.1.1 Challenges in SoC Security Verification

The prevalence of firmware in today's SoCs poses an important challenge in SoC security verification. Both firmware and hardware make many assumptions about the other component. Since functionality is implemented by a combination of hardware and firmware, verifying these two components separately requires explicitly enumerating these assumptions and ensuring the other component satisfies them. Bugs may exist in hardware or firmware alone, but some bugs may also be caused by incorrect assumptions made by hardware/firmware about the other component.

13.1.1.1 Need for Hardware/Firmware Co-verification

As an illustrative example, consider the runtime binary authentication protocol discussed by Krstic et al. [30] The objective of the protocol is to read a binary from an I/O device, verify it is signed by a trusted RSA public key, and if this is the case, load the binary into local memory for execution. Krstic et al. demonstrate that such a protocol is vulnerable to various attacks; e.g., a malicious entity may modify the loaded binary after its signature is verified, but before it is loaded for execution. To prevent this, firmware needs to configure the memory management unit (MMU) to "lock" the pages containing the binary during and after signature verification. Verifying that this protection works requires precise specification of the hardware/firmware interface for the MMU, ensuring that hardware correctly implements the protection such that untrusted hardware components cannot write to locked pages, and verifying that firmware sets the MMU configuration correctly. A mistake in any of these steps could violate the security requirements of the SoC.

The above example demonstrates: (1) the need for verification that analyzes the hardware and firmware together and (2) the need for precise specification of the hardware/firmware interface so that assumptions made by either side about the other are explicit. One way of achieving these two objectives is formal verification of firmware along with the cycle-accurate and bit-precise register transfer level (RTL) model of SoC hardware. Unfortunately, this naïve approach does not work in practice: formal verification of the RTL description along with the firmware is not feasible even for very small SoCs due to scalability limitations of formal tools.

13.1.1.2 SoC Verification Through Abstraction

A general technique for making SoC verification tractable is to use an abstraction that accurately models all updates to firmware-accessible hardware states [24, 33, 38, 49, 54, 55]. When verifying properties involving firmware, the abstraction is used instead of the bit-precise cycle-accurate hardware model.

Although the idea of constructing abstractions for firmware verification is attractive, there are several challenges in applying this technique. Firmware interacts with hardware components in a myriad of ways. For the abstraction to be useful, it needs to model all interactions and capture all updates to firmware-accessible states.

- Firmware usually controls accelerators in the SoC by writing to memory-mapped registers within the accelerators. These registers may set the mode of operation of the accelerator, the location of the data to be processed, or return the current state of the accelerator's operation. The abstraction needs to model these "special" reads and writes to the memory-mapped I/O space correctly.
- Once operation is initiated, the accelerators step through a high-level state machine that implements the data processing functionality. Transitions of this state machine may depend on responses from other SoC components, the

acquisition of semaphores, external inputs, etc. These state machines have to be modeled to ensure there are no bugs involving race conditions or malicious external input that cause unexpected transitions or deadlocks.
- Another concern is preventing compromised/malicious firmware from accessing sensitive data. To prove that such requirements are satisfied, the abstraction needs to capture issues such as a sensitive value being copied into a firmware-accessible temporary register.

Completeness of the abstraction is very important for security verification as finding security vulnerabilities requires reasoning about *all* inputs and states of the system including invalid/illegal inputs. A specific example of this is given in [48] which describes a bug affecting certain misaligned store instructions in a commercial SoC. A misaligned store instruction will cause an exception and therefore should not be executed by a well-behaved program. However, malicious code may specifically execute this instruction in order to exploit the bug and corrupt MMU state. If verification were limited to "legal" inputs and states, or if an abstraction did not precisely model the behavior under illegal inputs, such violations will be missed.

Manual construction of abstractions which capture these details, as proposed, for example, in [54, 55], is not practical because it is error-prone, tedious, and time-consuming. Manual construction can be especially challenging for third-party IPs because RTL "source code" may not be available so the abstraction has to be "reverse-engineered" from a gate-level netlist. If the manually constructed abstraction is *incorrect*, i.e., the hardware implementation is not consistent with the abstraction, properties proven using it are not valid.

13.1.1.3 Challenges in Specifying Security Properties

Another challenge in security verification is property specification. Traditional property specification languages based on temporal logic cannot express security requirements involving information flow: e.g., confidentiality and integrity [34]. As a result, existing formal tools such as model checkers and symbolic execution engines cannot verify information flow properties.

Confidentiality and integrity can be intuitively specified using *information flow* properties. One method for verifying these properties is through dynamic taint analysis [3, 13, 14, 29, 42, 46]. Dynamic taint analysis (DTA) associates a "taint bit" with each object in the program/design. Taint propagation rules then set the taint bit of the output of a computation if its input(s) are also tainted. Confidentiality violations are detected by tainting the secret and raising an error when the taint propagates to an untrusted object. Integrity violations are detected by tainting untrusted objects and raising an error when the taint propagates to a trusted location.

While DTA enables intuitive property specification, it has deficiencies in the verification aspect. DTA can determine if the property has been violated given an instruction trace. However, since it is a dynamic analysis, it cannot be used to

exhaustively search over the space of all possible instruction sequences to prove the absence of property violations. Secondly, creating taint propagation rules is challenging due to the problems of under- and over-tainting [3, 29]. Over-tainting can result in a deluge of false-positives [29] while under-tainting can miss bugs [42].

Static taint analysis can ensure information flow properties are satisfied in programs. However, current static taint analysis techniques are based on programming languages with secure type systems [37, 39]. These techniques cannot be applied on existing firmware because significant parts are written in assembly language. This necessitates analysis at the level of binary code rather than a high-level programming language. The above discussion points to the need for tools that can perform exhaustive analysis of information flow properties on binary code in the context of hardware/firmware co-verification.

13.1.2 SoC Security Verification Using Instruction-Level Abstractions

In this chapter, we introduce a principled methodology for the construction of abstractions for verification of system-level security properties in SoCs. The insight underlying our work is that firmware can only view changes in system state at the granularity of instructions. Therefore, it is sufficient to construct an abstraction which models hardware components of the SoC at this granularity. We call this an *instruction-level abstraction* (*ILA*) [49].

13.1.2.1 ILA Synthesis and Verification

To help easily construct the abstraction in a semi-automated manner, we build on recent progress in syntax-guided synthesis [2, 26, 27, 43]. Instead of manual construction of the complete ILA, the verification engineer constructs a *template* abstraction, which can be regarded as an abstraction with "holes." Our synthesis framework fills in the "holes" through directed simulation of hardware components.

The synthesis algorithm requires a simulator that can be used in the following manner: start simulation at a specified initial state, execute a given instruction, and return the state after the state update(s) corresponding to this instruction are carried out. This only requires minor modifications to an architectural, RTL, or gate-level simulator. The key advantages of ILA synthesis are that it significantly reduces manual effort in ILA construction and allows ILAs to be constructed easily and correctly even in the case of third-party IPs for which RTL descriptions may not be available.

The ILA can be verified to be a correct over-approximation of the hardware implementation. This uses model checking and ensures the ILA accurately captures the behavior of the RTL description. If the model checking is completed, we have a strong guarantee that all properties proven using the ILA are valid.

13.1.2.2 Security Verification Using the ILA

To address the problem of security property specification, we introduce a specification language for information flow properties of firmware. These properties specify that information cannot "flow" from a given source to a given destination and can be used to verify confidentiality when the source is a secret and the destination is any untrusted location. Integrity can be verified when the source is an untrusted location and the destination is a sensitive firmware register.

Using the ILA as a formal model of the underlying hardware, we introduce an algorithm based on symbolic execution to verify these information flow properties. The algorithm exhaustively explores all paths in the program and creates symbolic expressions of computation performed on each path. It then uses a constraint solver to check whether two different values at the source can result in different values at the destination. If yes, it means information flow can occur from source to destination as the destination value depends on the source. This means the property is violated. If no such value can be found, the property holds along this particular path.

13.1.2.3 Summarizing ILA-Based Verification

An overview of the complete methodology is shown in Fig. 13.2. The methodology has two parts: (a) synthesis and verification of correctness of the ILA and (b) using the ILA to verify security properties. The rest of this chapter is organized as follows. The definition of the ILA, ILA synthesis, and verification of ILA correctness are described in Sect. 13.2. The security property specification language and use of the ILA for verification properties is described in Sect. 13.3. A discussion

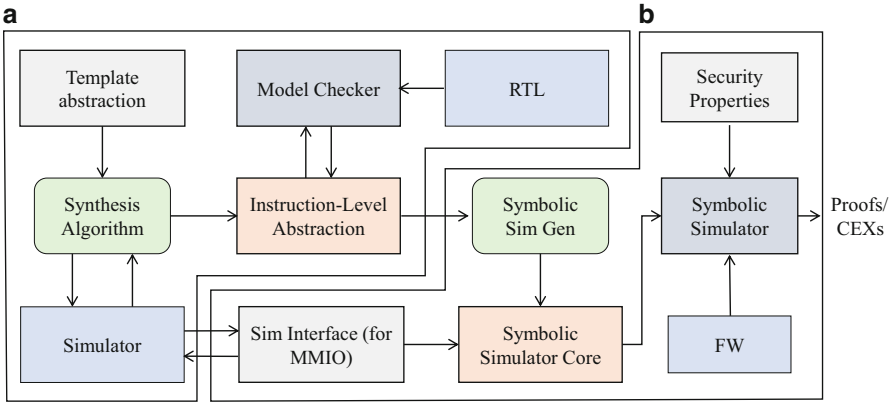


Fig. 13.2 Overview of ILA-based security verification methodology. (**a**) ILA synthesis and verification of correctness. (**b**) Security verification using ILAs

of potential extensions, limitations of the methodology, and related work is in Sect. 13.4. Section 13.5 provides concluding remarks.

13.2 Instruction-Level Abstractions

An instruction-level abstraction (ILA) is an abstraction that captures the firmware-visible hardware behavior in the context of an SoC. The key insight underlying the ILA is to model all firmware-visible state and associated state updates, and nothing more. Therefore, microarchitectural states, such as pipeline registers and reorder buffers, *are not* modeled in the ILA. However, all firmware-visible registers, including MMU state and on-chip network state *are* modeled in the ILA. This results in an abstraction which can be used for co-verification of hardware and firmware, but omits unnecessary detail that can cause scalability bottlenecks.

In the rest of this section, we first provide an intuitive overview of ILAs. We then formally define ILAs and describe how ILAs can be semi-automatically synthesized and verified to be correct abstractions of SoC hardware.

13.2.1 ILA Overview

Recall that the ILA is an abstraction of the firmware-visible behavior of a hardware module. Therefore, in the case of programmable cores, the ILA is the same as the instruction-set architecture (ISA) specification of the programmable core. An important aspect of the ILA that it is a formal, machine-readable abstraction that models all architectural state and associated state updates performed by the programmable core upon the execution of each instruction supported by it. Formal specifications of ISAs can be quite difficult to construct in practice [22, 23]. Our methodology mitigates this challenge by enabling *semi-automated synthesis* of ILAs.

The ILA models semi-programmable and fixed-function accelerators using an abstraction that is similar to an ISA. Accelerators in today's SoC designs are event-driven. Computation is performed in response to commands sent by programmable cores and is typically bounded in length. Our insight here is to view commands from the programmable cores to the accelerators as analogous to "instruction opcodes" and state updates in response to these commands as "instruction execution." The key insight here is to model accelerators using the same fetch/decode/execute sequence as a programmable core. The command is analogous to "fetch," the case-split determining how the command is processed is "decode," and the state update is instruction "execution."

The ILA models the effects of its abstracted "instructions" much like an ISA. It specifies the computation performed by each instruction and what updates to architectural state will occur. For ILAs, the architectural state now includes all

software-visible state that is accessible to its execution in the system, including memory-mapped registers, shared memory buffers, accelerator scratchpads, etc. For example, in the accelerator from [49] implementing the SHA-1 hashing algorithm, the `ComputeHash` instruction reads the source data from memory, computes its hash, and writes this result back to memory. The location of this instruction's input and output data in memory is set by previous configuration instructions.

Imposing this structure on an abstraction for accelerators has two advantages. First, the ILA becomes a precise and formal specification of the HW/SW interface for accelerators. It can be used in various design tasks such as full-system simulation to support FW/SW development and system-level verification. Second, interactions of FW with accelerators can be modeled using well-understood instruction-interleaving semantics, enabling use of standard tools such as model checkers for verification. Third, verification of SoC hardware also becomes more tractable because conformance with an ILA can be done compositionally on a "per-instruction" basis leveraging the body of work in microprocessor verification [28, 35].

13.2.2 ILA Definition

We now provide a formal definition of an instruction-level abstraction.

13.2.2.1 Notation

Let $\mathbb{B} = \{0, 1\}$ be the Boolean domain and let $bvec_l$ denote all bitvectors of width l. $M_{k \times l} : bvec_k \mapsto bvec_l$ maps from bitvectors of width k to bitvectors of width l and represents memories of address width k bits and data width l bits. Booleans and bitvectors are used to model state registers while the memory variables model hardware structures like scratchpads and random access memories. A memory variable supports two operations: $read(mem, a)$ returns data stored at address a in memory mem while $write(mem, a, d)$ returns a new memory which is identical to mem except that address a maps to d, i.e., $read(write(mem, a, d), a) = d$.

13.2.2.2 Architectural State and Inputs

The architectural state variables of an ILA are modeled as Boolean, bitvector, or memory variables. As with ISAs, the architectural state refers to state that is persistent and visible across instructions.

Let S represent the vector of state variables of an ILA consisting of Boolean, bitvector, and memory variables. In an ILA for a microprocessor, S contains all the architectural registers, bit-level state (e.g., status flags), and data and instruction

memories. In an accelerator, S contains all the software-visible registers and memory-structures. A state of an ILA is a valuation of the variables in S.

Let vector W represent the input variables of the ILA; these are Boolean and bitvector variables which model input ports of processors/accelerators.

Let $type_{S[i]}$ be the "type" of state variable $S[i]$; $type_{S[i]} = \mathbb{B}$ if $S[i]$ is Boolean, $type_{S[i]} = bvec_l$ if $S[i]$ is a bitvector of width l and $type_{S[i]} = M_{k\times l}$ if $S[i]$ is a memory.

13.2.2.3 Fetching an Instruction

The result of fetching an instruction is an "opcode." This is modeled by the function $F_o : (S \times W) \mapsto bvec_w$, where w is the width of the opcode. For instance, in the 8051 microcontroller, $F_o(S, W) \triangleq read(\text{IMEM}, \text{PC})$ where IMEM is the instruction memory and PC is the program counter.[1]

Programmable cores repeatedly fetch, decode, and execute instructions, i.e., they always "have" an instruction to execute. However, accelerators may be event-driven and execute an instruction only when a certain trigger occurs. This is modeled by the function $F_v : (S \times W) \mapsto \mathbb{B}$. Suppose an accelerator executes an instruction when either Cmd1Valid or Cmd2Valid is asserted, then $F_v(S, W) \triangleq$ Cmd1Valid $\vee$ Cmd2Valid.

13.2.2.4 Decoding an Instruction

Decoding an instruction involves examining an opcode and choosing the state update operation that will be performed. We represent the different choices by defining a set of functions $D = \{\delta_j \mid 1 \leq j \leq C\}$ for some constant C where each $\delta_j : bvec_w \mapsto \mathbb{B}$. Recall $F_o : (S \times W) \mapsto bvec_w$ is a function that returns the current opcode. Each δ_j is applied on the result of F_o. The functions δ_j must satisfy the condition: $\forall j, j' : j \neq j' \iff \neg(\delta_j \wedge \delta_{j'})$.

For convenience let us also define the predicate $\text{op}_j \triangleq \delta_j(F_o(S, W))$. When op_j is 1, it selects the jth instruction. For example, in the case of the 8051 microcontroller, $D = \{\delta_1(f) \triangleq f = 0, \delta_2(f) \triangleq f = 1, \ldots, \delta_{256}(f) \triangleq f = 255\}$.[2] Recall we had defined F_o for this microcontroller as $F_o(S, W) \triangleq read(\text{IMEM}, \text{PC})$. Therefore, $\text{op}_j \iff read(\text{IMEM}, \text{PC}) = j - 1$. We are "case-splitting" on each of the 256 values taken by the opcode and each of these possibly performs a different state update. The functions δ_j choose which of these updates is to be performed.

[1] Note $F_o(S, W)$ must contain the instruction opcode. It may also (but is not required to) contain additional data such as the arguments to the instruction.

[2] We are writing bitvectors of width 8 (elements of $bvec_8$) as $0 \ldots 255$.

13.2.2.5 Executing an Instruction

For each state element $S[i]$ define the function $N_j[i] : (S \times W) \mapsto type_{S[i]}$. $N_j[i]$ is the state update function for $S[i]$ when $\text{op}_j = 1$. For example, in the 8051 microcontroller, opcode `0x4` increments the accumulator. Therefore, $N_4[\text{ACC}] = \text{ACC} + 1$. The complete next state function $N : (S \times W) \mapsto S$ is defined in terms of the functions $N_j[i]$ over all i and j.

Defining the next state function N compositionally in terms of the functions $N_j[i]$ has an important advantage: it enables compositional verification [28, 35]. The behavior of the RTL can now be verified separately for each state element $S[i]$ and for each opcode op_j. This results in a simpler verification problem and enables scalable verification of large designs.

13.2.2.6 Syntax

The language of expressions allowed in F_o, F_v, D, and $N_j[i]$ is shown in Fig. 13.3. Expressions are quantifier-free formulas over the theories of bitvectors, bitvector arrays, and uninterpreted functions. They can be of type Boolean, bitvector, or memory, and each of these has the usual variables, constants, and operators with

⟨*exp*⟩ ::= ⟨*bv-exp*⟩ | ⟨*bool-exp*⟩ | ⟨*mem-exp*⟩ | ⟨*func-exp*⟩
| ⟨*choice-exp*⟩

⟨*bv-exp*⟩ ::= var ⟨*id*⟩ *width* | cnst `val` *width*
| bvop ⟨*exp*⟩ ...
| read ⟨*memexp*⟩ ⟨*addrexp*⟩
| read-block ⟨*memexp*⟩ ⟨*addrexp*⟩
| apply ⟨*func*⟩ ⟨*bv-exp*⟩ ...
| **extract-bitslice** ⟨*bv-exp*⟩ *width*
| **extract-subword** ⟨*bv-exp*⟩ *width*
| **replace-bitslice** ⟨*bv-exp*⟩ ⟨*bv-exp*⟩
| **replace-subword** ⟨*bv-exp*⟩ ⟨*bv-exp*⟩
| **in-range** ⟨*bv-exp*⟩ ⟨*bv-exp*⟩

⟨*bool-exp*⟩ ::= var | true | false
| boolop ⟨*exp*⟩ ...

⟨*mem-exp*⟩ ::= ⟨*id*⟩ |
| write ⟨*mem-exp*⟩ ⟨*bv-exp*⟩ ⟨*bv-exp*⟩
| write-block ⟨*mem-exp*⟩ ⟨*bv-exp*⟩ ⟨*bv-exp*⟩

⟨*func-exp*⟩ ::= func ⟨*id*⟩ $width_{out}$ $width_{in_1}$ ...

⟨*choice-exp*⟩ ::= **choice** ⟨*exp*⟩ ⟨*exp*⟩ ...

Fig. 13.3 Syntax for expressions

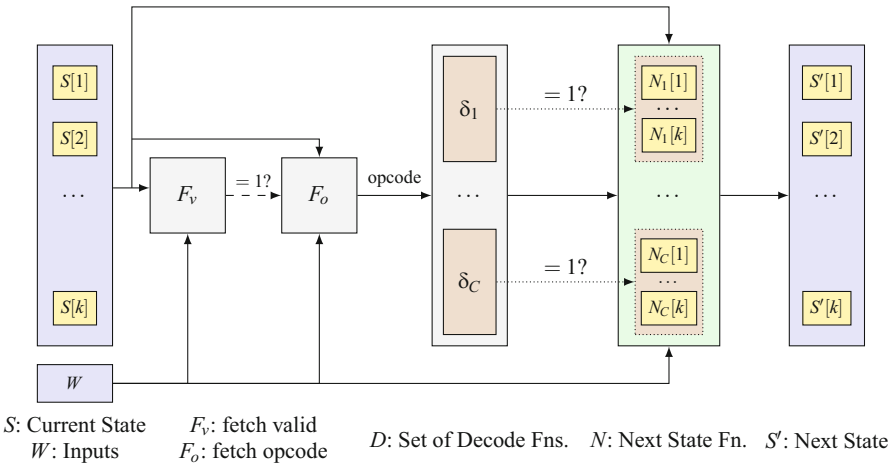


Fig. 13.4 Pictorial overview of the ILA definition

standard interpretations. The synthesis primitives, shown in **bold**, will be described in Sect. 13.2.3.

13.2.2.7 Putting It All Together

To summarize, an instruction-level abstraction (ILA) is the tuple: A $= \langle S, W, F_o, F_v, D, N \rangle$. S and W are the state and input variables. F_o, F_v, D, and N are the fetch, decode, and next state functions, respectively.

A pictorial overview of this definition is shown in Fig. 13.4. The leftmost box shows the state vector S. The function F_v ("fetch valid") examines the state vector S and input vector W and determines if there is an instruction to execute. If this is the case, i.e., if $F_v(S, W) = 1$, then $F_o(S)$ is used to get the opcode from the current state and input vectors. This opcode is used to evaluate each of the functions $\delta_1, \delta_2, \ldots, \delta_j, \ldots, \delta_C$. If $\delta_j = 1$, then $N_j[i]$ is evaluated for each $S[i]$ to get the next state $S'[i]$. The vector S' then defines the next state of the ILA and the above process is repeated again from this state to execute the next "instruction."

13.2.3 ILA Synthesis

For ILAs to be useful, one must be able to generate them correctly and preferably automatically. Due to the prevalence of third-party IPs, SoC hardware blocks often exist *before* the ILA is constructed, and so ILAs need to be constructed *post hoc* without IP designer support. To address the error-prone and tedious aspects of this construction, we have developed an algorithm for *template-based synthesis* of ILAs.

To semi-automatically synthesize the functions $N_j[i]$ in the ILA, we build on the counterexample guided inductive synthesis (CEGIS) algorithm [26, 27]. We assume availability of a simulator that models state updates performed by the hardware. This simulator can be gate-level, RTL, or a high-level C/C++/SystemC simulator of the design; it is only used as a black-box. Its implementation does not matter except that it be possible to set the initial state, execute an instruction, and read out the final state after execution.

13.2.3.1 Notation and Problem Statement

Let $Sim : (S \times W) \mapsto S$ be the I/O oracle for the next state function N. Define Sim_i as the function that projects the state element $S[i]$ from Sim; $Sim_i : (S \times W) \mapsto type_{S[i]}$. In order to help synthesize the function implemented by Sim_i, the SoC designers write a *template next state function*, denoted by $\mathscr{T}_i : (\Phi \times S \times W) \mapsto type_{S[i]}$.

Φ is a set of *synthesis* variables, also referred to as "holes" [45], and different assignments to (interpretations of) Φ result in different next state functions. Unlike $N[i]$, $\mathscr{T}_i$ is a partial description and is therefore easier to write. It can omit certain low-level details, such as the mapping between individual opcodes and operations, opcode bits and source and destination registers, etc. These details are filled-in by the counterexample guided inductive synthesis (CEGIS) algorithm [26, 27] by observing the output of Sim_i for carefully selected values of (S, W).

(Problem Statement: ILA Synthesis): For each state element $S[i]$ and each op_j, find an interpretation of Φ, $\Phi_j[i]$, such that $\forall S, W : op_j \implies (\mathscr{T}_i(\Phi_j[i], S, W) = Sim_i(S, W)))$.

The synthesis procedure is repeated for each instruction (each j) and each state element (each i), and the synthesis result $\Phi_j[i]$ is an interpretation of Φ for i and j.

13.2.3.2 Template Language

The synthesis primitives in the currently implemented template language are shown in bold in Fig. 13.3. It is important to emphasize that our algorithm and methodology

are not dependent on the specific synthesis primitives used. The only requirement placed on the primitives is that it be possible to "compile" them to some quantifier-free formula over bitvectors, bitvector arrays, and uninterpreted functions.

Synthesis Primitives

The expression **choice** ε_1 ε_2 asks the synthesizer to replace the **choice** primitive with either ε_1 or ε_2 based on simulation result. **choice** ε_1 ε_2 is translated to the formula $\text{ITE}(\phi_b, \varepsilon_1, \varepsilon_2)$ where $\phi_b \in \Phi$ is a new Boolean variable associated with this instance of the choice primitive. Its value is determined by the synthesis procedure.

The primitives **extract-slice** and **extract-subword** synthesize bitvector extract operators. The synthesis process determines the indices to be extracted from. These operators are used to extract bit-fields from a register. The advantage of using these operators instead of an extract operator is that the indices of the bitfield need not be specified. This is advantageous for two reasons: (1) it reduces manual effort when constructing the ILA and (2) it also eliminates errors in specifying indices, as these are now automatically determined by the synthesis procedure. The **replace-slice** and **replace-subword** are the counterparts of these primitives; they replace a part of the bitvector with an argument expression.

The primitive **in-range** synthesizes a bitvector constant that is within the specified range. We note that adding new synthesis primitives is easy and straightforward in our language.

13.2.3.3 An Illustrative Example

We illustrate the definition of an ILA and template next state function using the processor shown in Fig. 13.5. The instruction to be executed is read from the ROM. Its operands can either be an immediate from the ROM or from the 4-entry register file. For simplicity, we assume that the only two operations supported by the processor are addition and subtraction.

The architectural state for the processor is $S = \{\text{ROM}, \text{PC}, \text{R}_0, \text{R}_1, \text{R}_2, \text{R}_3\}$ and input set W is empty. $type_{\text{ROM}} = M_{8\times8}$ while all the other variables are of type

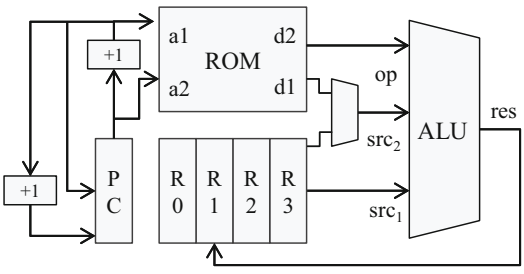


Fig. 13.5 A simple processor for illustration

$bvec_8$. The opcode which determines the next instruction is stored in the ROM and so $F_o \triangleq read(\text{ROM}, \text{PC})$; $F_v \triangleq true$. $D = \{\delta_j \mid 1 \leq j \leq 256\}$ where each $\delta_j(f) \triangleq f = j - 1$. The template next state functions, $\mathscr{T}_{\text{PC}}$ and $\mathscr{T}_{\text{R}_\text{i}}$, are as follows.

$$
\begin{aligned}
\mathscr{T}_{\text{PC}} &= \textbf{choice}\ (\text{PC} + 1)\ (\text{PC} + 2) \\
imm &= read(\text{ROM}, \text{PC} + 1) \\
src_1 &= \textbf{choice}\ \text{R}_0\ \text{R}_1\ \text{R}_2\ \text{R}_3 \\
src_2 &= \textbf{choice}\ \text{R}_0\ \text{R}_1\ \text{R}_2\ \text{R}_3\ imm \\
res &= \textbf{choice}\ (src_1 + src_2)\ (src_1 - src_2) \\
\mathscr{T}_{\text{R}_\text{i}} &= \textbf{choice}\ res\ R_i \quad (0 \leq i \leq 3)
\end{aligned}
$$

13.2.3.4 Synthesis Algorithm

The counterexample-guided inductive synthesis (CEGIS) algorithm from [49] is shown in Algorithm 1. The inputs to the algorithm are the following.

1. op_j which determines the opcode for which synthesis is performed,
2. The template next state function $\mathscr{T}_i$.
3. The simulator Sim_i. The suffix i here denotes that the value of state element $S[i]$ is extracted from the output of the simulator.
4. A is a conjunction of assumptions under which synthesis is to be carried out. For example, suppose the opcode is $read(\text{ROM}, \text{PC})$, and functionality is only defined for opcodes in the range `0x0` to `0xF0`, then the assumption A would express this as $A \triangleq read(\text{ROM}, \text{PC}) \geq \texttt{0x0} \wedge read(\text{ROM}, \text{PC}) \leq \texttt{0xF0}$.

The algorithm is executed for each op_j and each $S[i]$. In each case it returns $\Phi_j[i]$ which is used to compute the next state function as $N_j[i](S, W) = \mathscr{T}_i(\Phi_j[i], S, W)$.

Algorithm 1 Synthesis algorithm

```
1: function SynCEGIS(op_j, T_i, Sim_i, A)
2:     k ← 1
3:     R_1 ← A ∧ op_j ∧ (θ ↔ (T_i(Φ_1, S, W) ≠ T_i(Φ_2, S, W)))
4:     while sat(R_k ∧ θ) do
5:         Δ ← Model_(S,W)(R_k ∧ θ)                    ▷ get dist. input Δ
6:         O ← Sim_i(Δ)                                ▷ simulate Δ
7:         O_1 ← (T_i(Φ_1, Δ) = O)
8:         O_2 ← (T_i(Φ_2, Δ) = O)
9:         R_{k+1} ← R_k ∧ O_1 ∧ O_2                   ▷ enforce output O for Δ
10:        k ← k + 1
11:    end while
12:    if sat(R_k ∧ ¬θ) then
13:        return Model_{Φ_1}(R_k ∧ ¬θ)
14:    end if
15:    return ⊥
16: end function
```

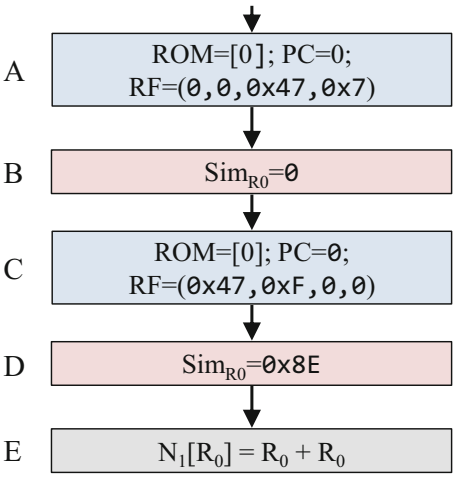


Fig. 13.6 Distinguishing inputs for the illustrative example. Notation *ROM* = [0] means all ROM addresses contain the value 0

To understand the algorithm, observe that $\mathscr{T}_i(\Phi, S, W)$ defines a *family* of next state functions. Different functions are selected by different interpretations of Φ. The algorithm tries to find an interpretation of (S, W), say Δ, which for some two interpretations of Φ, $\Phi_1[i]$ and $\Phi_2[i]$ is such that $\mathscr{T}_i(\Phi_1[i], \Delta) \neq \mathscr{T}_i(\Phi_2[i], \Delta)$. Δ is called a *distinguishing input*. Once Δ is found, the simulation oracle Sim_i can be evaluated to find the expected output for this scenario. We then add constraints enforcing this input/output relation and try to find another distinguishing input. This process repeats until no more distinguishing inputs can be found. When this happens, it means that all remaining interpretations of Φ result in the same next state function, and any such interpretation is the solution.

Consider the computation of $N_1[\mathrm{R}_0]$ shown in Fig. 13.6. We are constructing the next state function for R_0 for opcode $\mathrm{op}_1 \iff \mathit{read}(\mathrm{ROM}, \mathrm{PC}) = \texttt{0x0}$. The first distinguishing input computed is node A in Fig. 13.6. This distinguishes between next state functions like $\mathrm{R}_0 + \mathrm{R}_2$, $\mathrm{R}_0 + \mathrm{R}_3$ and functions like $\mathrm{R}_0 + \mathrm{R}_0$. Node B shows the output from the simulator for A is 0x0 ruling out $\mathrm{R}_0 + \mathrm{R}_2$ and $\mathrm{R}_0 + \mathrm{R}_3$. We now compute distinguishing input C. This input distinguishes among the functions $\mathrm{R}_0 + \mathrm{R}_0$, $\mathrm{R}_0 - \mathrm{R}_0$, $\mathrm{R}_0 + \mathrm{R}_1$, $\mathrm{R}_0 - \mathrm{R}_1$, and so on. When this input is simulated, it results in output D. At this point, we have a unique next state function: $N_1[\mathrm{R}_0] = \mathrm{R}_0 + \mathrm{R}_0$ and the algorithm terminates.

13.2.4 ILA Verification

Once we have ILA, the next step is to verify that it correctly abstracts the hardware implementation. Our first attempt at this might be to consider the ILA and the RTL as finite state transition systems executing in parallel and verify properties of the

form $\mathbf{G}(x_{\text{ILA}} = x_{\text{RTL}})$. This property says that state variable x_{ILA} in the ILA is equal in every time step[3] to the state variable x_{RTL} in the RTL. Unfortunately, this property is likely to be false for most state variables in hardware designs.

For example, consider a pipelined microprocessor with branch prediction. In this case, the processor may mispredict a branch and execute "wrong-path" instructions. Although these instructions will eventually be flushed, while they are being executed registers in the RTL will contain the results of these "wrong-path" instructions and so x_{RTL} will not match x_{ILA}.

13.2.4.1 Verifying Abstraction Correctness

When considering the internal state of the hardware components, we verify the ILA by defining *refinement relations* as proposed by McMillan [35]: $\mathbf{G}(\text{cond}_{ij} \implies x_{\text{ILA}} = x_{\text{RTL}})$. The predicate cond_{ij} specifies when the equivalence between state in the ILA and the corresponding state in the implementation holds. For example, in a pipelined microprocessor, we might expect that when an instruction commits, the architectural state of the implementation matches the ILA. Note the suffix *ij* here denotes that the refinement relation may be different for each state element $S[i]$ and each opcode op_j, i.e., the state update of the RTL may occur in a different cycle for each combination of $S[i]$ and op_j.

Defining the refinement relations as above allows compositional verification [28]. Consider the property $\neg(\phi\ \mathbf{U}\ (\text{cond}_{ij} \wedge (x_{\text{ILA}} \neq x_{\text{RTL}})))$ where ϕ states that all refinement relations hold until time $t-1$. This is equivalent to the above property, but we can abstract away irrelevant parts of ϕ when proving equivalence of x_{ILA} and x_{RTL}. For example, when considering op_j, we can abstract away the implementation of other opcodes $\text{op}_{j'}$ and assume these are implemented correctly. This simplifies the model but these proofs are still valid because we are still considering the correctness of all opcodes, albeit separately.

For state variables that are outputs of hardware components being modeled, we expect that ILA outputs always match the implementation. In this case, the property is $\mathbf{G}(x_{\text{ILA}} = x_{\text{RTL}})$.

13.2.4.2 Discussion of Verification Issues

One part of our case study is a pipelined microcontroller with limited speculative execution. Our refinement relations are of the form $\mathbf{G}(\textit{inst\_finished} \implies (x_{\text{ILA}} = x_{\text{RTL}}))$, i.e., the state of the ILA and implementation must match when each instruction commits. The other part involves the verification of two cryptographic accelerators. Here the refinement relations are of the following form: $\mathbf{G}(\textit{hlsm\_state\_changed} \implies x_{\text{ILA}} = x_{\text{RTL}})$. The predicate *hlsm_state_changed* is

[3] This is the meaning of the **G** linear temporal logic (LTL) operator.

equal to 1 whenever the high-level state machine in the accelerator changes state. This refinement relation states that the high-level state machines of the ILA and RTL have the same transitions. The RTL state machine has some "low-level" states. These states do not exist in the ILA and are not visible to the firmware.

If we had to verify a superscalar processor, the ILA would execute multiple instructions in each transition. The exact number of instructions to be executed with each transition is an output of the implementation and an input to the abstraction. The property would state that after these many instructions are executed, the states of the ILA and implementation match.

Propagating auxiliary execution information from the implementation to the abstraction can help verify many complex scenarios. Consider the verification of weak memory models [1] where loads and stores in a microprocessor can be reordered in (well-defined) ways that affect program semantics. The information about the order in which load/store instructions are executed will be an output of the implementation and an input to the ILA. The refinement relations would verify that the execution of the ILA matches the implementation assuming instructions are executed in the same order.

13.2.4.3 Verification Correctness

If we prove the refinement relations for all outputs of the ILA and implementation: $\mathbf{G}(x_{\text{ILA}} = x_{\text{RTL}})$, then we know that the ILA and implementation have identical externally visible behavior. Hence any properties proven about the behavior of the external inputs and outputs of the ILA are also valid for the implementation.

In practice, proving the property $\mathbf{G}(x_{\text{ILA}} = x_{\text{RTL}})$ for all external outputs may not be scalable, so we adopt McMillan's compositional approach. We prove refinement relations of the form $\neg(\phi\ \mathbf{U}\ (\text{cond}_{ij} \wedge x_{\text{ILA}} \neq x_{\text{RTL}}))$ for internal state and use these to prove the equivalence of the outputs.

If these compositional refinement relations are proven for all firmware-visible state in the ILA and implementation, then we know that all firmware-visible state updates are equivalent between the ILA and the implementation. Further, we know that transitions of the high-level of state machines in the ILA are equivalent to those in the implementation. These properties guarantee that firmware/hardware interactions in the ILA are equivalent to the implementation, ensuring correctness of the abstraction.

13.2.5 Practical Case Study

This section describes the evaluation methodology, the example SoC used as a case study, and then briefly describes the synthesis and verification results.

13.2.5.1 Methodology

The template-based synthesis framework was implemented as a Python library using the Z3 SMT solver [16]. A modified version of Yosys was used to synthesize netlists from behavioral Verilog [53]. We used ABC for model checking [5]. The synthesis framework, template abstractions, synthesized ILA, and other experimental artifacts are available online [19].

13.2.5.2 Example SoC Structure

Experiments were conducted on an example SoC constructed from open source components containing the 8051 microcontroller and two cryptographic accelerators (Fig. 13.7). One accelerator implements encryption/decryption using the Advanced Encryption Standard (AES) while the other implements the SHA-1 cryptographic hash function [25, 47]. The RTL description of the 8051 is from OpenCores.org [51]. We also used *i8051sim* for instruction-level simulations of the 8051 [32].[4]

The firmware running on the 8051 initiates operation of the accelerators by writing the addresses of the data to be encrypted/decrypted/hashed to memory-mapped registers within the accelerators. Operation is started by writing to the start register which is also memory-mapped. Once the operation is started, the accelerators use direct memory access (DMA) to fetch the data from the external memory (XRAM), perform the operation, and write the result back to XRAM. The processor determines completion by polling a memory-mapped status register.

13.2.5.3 Summary of Synthesis Results

We constructed two ILAs: one for the 8051 microcontroller and another for the arbiter, XRAM, AES, and SHA modules. The insight here is that the 8051 communicates with the accelerators and XRAM by reading/writing to XRAM addresses. So from the perspective of the 8051, it is sufficient to show that all instructions that

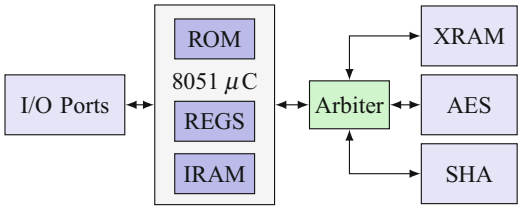


Fig. 13.7 Example SoC block diagram

[4] We could have used the Verilog implementation itself for simulation. However, we chose to replicate the common scenario where simulators are developed and used for validation before the RTL design is complete.

modify the internal state of the 8051 are executed correctly and instructions which read/write XRAM produce the correct results at the external memory interface. What happens after these instructions "leave" the external memory interface—whether they modify the XRAM or start AES encryption, or return the current state of the SHA accelerator—need not be considered in this model. For the accelerators and the XRAM, we construct a separate ILA and the only instructions we need to consider here are reads and writes to XRAM addresses. In this ILA, we verify that these operations produce the expected results.

As an indication of the effort involved in building the model, the size of the template ILA, the simulator, and the RTL implementations for the two ILAs is shown in Table 13.1. The table shows that the template ILA is typically much smaller than the RTL. These numbers also demonstrate the template ILA can be written with relatively less effort.

Execution times of the synthesis algorithm for the 8051 ILA are shown in Table 13.2. We report the average and maximum values over all 256 opcodes. Except for the internal RAM, all other elements are synthesized with a few seconds. ILA synthesis found five bugs in *i8051sim*.

13.2.5.4 Typical ILAs

Tables 13.3 and 13.4 show "instructions" in the ILAs for the accelerators. Firmware first *configures* the accelerator with the appropriate instructions and then starts operation with the `StartEncryption` and `StartHash` instructions. It can then poll for completion using the `GetStatus` instruction. Each of these instructions

Table 13.1 Lines of code (LoC) and size in bytes of each model for the 8051 ILA

	8051		AES+SHA+XRAM	
Model	LoC	Size (KB)	LoC	Size (KB)
Template ILA	≈ 650	30	≈ 500	26
Instruction-level simulator	≈ 3000	106	≈ 400	14
Behavioral verilog implementation	≈ 9600	360	≈ 2800	87

Table 13.2 Synthesis execution time for 8051 ILA

	AVG	MAX		AVG	MAX
State	Time (s)		State	Time (s)	
ACC	4.3	8.5	B	3.6	5.1
DPH	2.7	5.0	DPL	2.6	4.4
IRAM	1245.7	14043.6	P0	1.8	2.7
P1	2.4	3.8	P2	2.2	3.5
P3	2.7	4.6	PC	6.3	141.2
PSW	7.3	15.9	SP	2.8	5.0
XRAM/addr	0.4	0.4	XRAM/dataout	0.3	0.4

Table 13.3 ILA instructions for AES accelerator

Instruction	Description of operation
`Rd/Wr DataAddr`	Get/set the address of data to encrypt
`Rd/Wr DataLen`	Get/set the length of data to encrypt
`Rd/Wr Key0 <index>`	Get/set specified byte of key0
`Rd/Wr Key1 <index>`	Get/set specified byte of key1
`Rd/Wr Ctr <index>`	Get/set specified byte of counter
`Rd/Wr KeySel`	Get/set the current key (key0/key1)
`StartEncryption`	Start the encryption state machine
`GetStatus`	Poll for completion

Table 13.4 ILA instructions for SHA1 accelerator

Instruction	Description of operation
`Rd/Wr DataInputAddr`	Get/set address of data to be hashed
`Rd/Wr DataLength`	Get/set length of data to be hashed
`Rd/Wr DataOuptutAddr`	Get/set the address of output
`StartHash`	Start the SHA1 state machine
`GetStatus`	Poll for completion

is "triggered" by a memory-mapped I/O write to an appropriate address; F_v is of following form $F_v \triangleq ((\text{memop} = \texttt{WR} \vee \text{memop} = \texttt{RD}) \wedge \text{mem\_addr} = \texttt{REG\_ADDR})$ where `REG_ADDR` is the address of the corresponding configuration register.

13.2.5.5 Summary of Verification Results

We generated Verilog "golden models" from the ILAs and defined a set of refinement relations specifying that the golden models were equivalent to the RTL. We then used bounded and unbounded model checking to verify these refinement relations. These verification results are described below.

Verification of the 8051 ILA

We first attempted to verify the 8051 by generating a large monolithic golden model that implemented the entire functionality of the processor in a single cycle. The IRAM in this model was abstracted from a size of 256 bytes to 16 bytes. This abstracted golden model was generated automatically using the synthesis library. We manually implemented the abstraction reducing the size of the IRAM in the RTL implementation.

We used this golden model to verify properties of the form $\mathbf{G}(\textit{inst\_finished} \implies x_{\text{ILA}} = x_{\text{RTL}})$. For the external outputs of the processor, e.g., the external RAM address and data outputs, the properties were of the form $\mathbf{G}(\textit{output\_valid} \implies$

Table 13.5 Results with per-instruction golden model

	BMC bounds					
Property	CEX	≤ 20	≤ 25	≤ 30	≤ 35	Proofs
PC	0	0	25	10	204	96
ACC	1	0	8	39	191	56
IRAM	0	0	10	36	193	1
XRAM/dataout	0	0	0	0	239	238
XRAM/addr	0	0	0	0	239	239

$x_{\text{ILA}} = x_{\text{RTL}}$). Verification was done using bounded model checking (BMC) with ABC using the `bmc3` command. After fixing some bugs and disabling the remaining (17) buggy instructions, we were able to reach a bound of 17 cycles after 5 h of execution.

To improve scalability, we generated a set of "per-instruction" golden models which only implement the state updates for one of the 256 opcodes, the implementation of the other 255 opcodes is abstracted away. We then verified a set of properties of the form: $\neg(\phi\ \mathbf{U}(\mathit{inst\_finished} \wedge \text{op}_j \wedge x_{\text{ILA}} \neq x_{\text{RTL}}))$. Here ϕ states that all architectural state matches until time $t-1$. We then attempted to verify five important properties stating that: (1) PC, (2) accumulator, (3) the IRAM, (4) XRAM data output, and (5) XRAM address must be equal for the golden model and the implementation.

Results for these verification experiments are shown in Table 13.5. Each row of the table corresponds to a particular property. Columns 2–6 show the bounds reached by BMC within 2000 s. For example, the first row shows that for 25 instructions, the BMC was able to reach a bound between 21 and 25 cycles without a counterexample; for 10 instructions, it achieved a bound between 26 and 30 cycles, and for the remaining 204 instructions, BMC reached a bound between 31 and 35 cycles. The last column shows the number of instructions for which we could prove the property. These proofs were done using the `pdr` command which implements the IC3 algorithm [8] with a time limit of 1950 s. Before running `pdr`, we preprocessed the netlists using the gate-level abstraction [36] technique with a time limit of 450 s.

Bugs Found During 8051 Verification

In the simulator, we found five bugs in total. Bugs in `CJNE`, `DA`, and `DIV` instructions were due to signed integers being used where unsigned values were expected. Another was a typo in `AJMP` and the last was a mismatch between RTL and the simulator when dividing by zero. These bugs were found during synthesis.

An interesting bug in the template was for the `POP` instruction. The `POP <operand>` instruction updates two items of state: (1) `<operand> = RAM[SP]` and (2) `SP = SP - 1`. But what if operand is `SP`? The RTL set `SP` using (1) while the ILA used (2). This was discovered during model checking and the ILA was changed to match the RTL. This shows one of the benefits of our methodology: all state updates are precisely defined and consistent between the ILA and RTL.

In the RTL model, we found a total of 7+1 bugs. One of these is an entire class of bugs related to the forwarding of special function register (SFR) values from an in-flight instruction to its successor. This affects 17 different instructions and all bit-addressable architectural state. We partially fixed this. A complete fix appears to require significant effort.

Another interesting issue was due to reads from reserved/undefined SFR addresses. The RTL returned the previous value stored in a temporary buffer. This is an example of the methodology detecting and preventing unintended leakage of information through undefined state.

Verifying the XRAM+AES+SHA ILA

We generated a Verilog golden model that combined the ILAs for the XRAM, AES, and SHA modules. We reduced the size of the XRAM in the ILA and the implementation to just one byte because we were not looking to prove correctness of reads and writes to the XRAM. We then attempted to prove a set of properties of the form $\mathbf{G}(\mathit{hlsm\_state\_change} \implies (x_{\mathrm{ILA}} = x_{\mathrm{RTL}}))$. We were able to prove that the *AES:State*, *AES:Addr*, and *AES:Len* in the implementation matched the ILA using the `pdr` command. For other firmware-visible state, BMC found no property violation up to 199 cycles with a time limit of 1 h.

13.3 Security Verification Using ILAs

The ILA is a complete formal specification of hardware behavior and enables scalable system-level verification. Firmware that interacts with accelerators and I/O devices can now be analyzed in terms of firmware-visible behavior as specified by the ILA instead of ad hoc manually constructed models, or at the other extreme, very detailed bit-precise cycle-accurate RTL descriptions.

In this section, we describe how ILAs can be used to verify confidentiality and integrity properties of firmware using symbolic execution. We first describe the system and threat model. We then describe a specification language for firmware security properties and briefly provide an overview of an algorithm based on symbolic execution for verifying these properties.

13.3.1 System and Threat Model Overview

This section provides a brief overview of the system model and threat model considered in this work.

13.3.1.1 System-On-Chip Model

As described in Sect. 13.1 and shown in Fig. 13.1, we consider SoC designs consisting of a set of interacting *IPs*, a shared I/O space, an on-chip interconnect and possibly a shared memory. Each IP consists of a microcontroller, specialized hardware components, and private read-only and read-write memories (i.e., ROMs and RAMs). The firmware executes on the microcontroller interacts with the specialized hardware in its own IP as well as other IPs through memory-mapped I/O (MMIO).

The firmware's view of the system consists of a set of architectural registers, a special register known as the program counter, an instruction ROM which contains the firmware, and a data RAM which is used by the firmware during execution. We consider single-threaded firmware, however, other hardware and firmware interactions are also modeled and execute concurrently with the firmware thread. We use the ILA of the microcontroller to model the state updates performed by each instruction in the firmware.

13.3.1.2 Threat Model

The verification problem is tackled in a modular manner. Each IP is verified separately and the threat model is defined from the perspective of the individual IPs.

For each IP we identify the other IPs which are its *trust boundary*. The trust boundary is the subset of other IPs this IP fully trusts. For example, an IP involved in security critical functionality such as secure boot will likely *not trust* the camera and GPS IPs. Therefore, these IPs *will be outside* its trust boundary and any inputs it receives from these IPs are untrusted. Keeping the trust boundary small helps keep the attack surface small.

13.3.1.3 Security Objectives

The two classes of security objectives we consider are *confidentiality* and *integrity* of firmware assets. We wish to keep firmware secrets, e.g., encryption keys, confidential from untrusted IPs. Similarly, we wish to preserve the integrity of firmware assets. For example, we wish to ensure the integrity of firmware control-flow by ruling out stack smashing/buffer overflow attacks in the presence of arbitrary

inputs from untrusted IPs. We also wish to ensure that untrusted IPs cannot modify the value of sensitive control/configuration registers, such as memory protection configuration registers. In this work, we do not consider other security requirements like availability and side channel attacks.

13.3.1.4 Modelling the Attacker

We assume that memory, I/O, and hardware registers controlled by untrusted IPs contain arbitrary values. In other words, outputs of an untrusted IP are *unconstrained* so reads from these locations return arbitrary values. Similarly, untrusted IPs may send invalid commands in an attempt to exploit, for example, buffer overflow bugs.

13.3.2 Specifying Information Flow Properties

The property specification language for firmware security properties is based on the following insights. First, security requirements such as confidentiality and integrity are essentially statements about information flow. These express the requirement that either a firmware secret must not "flow" to an untrusted value (confidentiality), or an untrusted value must not "flow" to a sensitive asset (integrity). Note such properties cannot be expressed using specification languages based on temporal logic [34].

Second, almost all interesting firmware security assets, such as secret keys, sensitive configuration registers, and untrusted input registers, are accessed through memory-mapped I/O (MMIO). Therefore, firmware address ranges and architectural registers are first class entities in the property specification language.

Third, a mechanism for *declassification* is required [40]. This allows information flow when certain conditions hold during execution; information flow is disallowed if these do not hold.

Based on the above requirements, we introduce a specification language for information flow properties consisting of:

1. An `src` which is a range of firmware memory addresses.
2. A predicate `srcpred` associated with the source which specifies when the data at `src` is valid. For example, we may allow a register to be programmed from an input port during the boot process, but not afterwards with the predicate $\neg boot$.
3. A `dst` which is an element of the ILA state.

(continued)

4. A predicate `dstpred` for `dst` which specifies when data at `dst` is valid similar to (2) above.

The property holds if data read from `src` when `srcpred=1` never influences a value written to `dst` when `dstpred=1`. `srcpred` and `dstpred` are evaluated at the time of the read and write, respectively.

13.3.3 Firmware Execution Model

We now describe firmware state and the execution model.

13.3.3.1 Execution State

Firmware state is modeled using an ILA of the microcontroller hardware: $A = \langle S, W, F_o, F_v, D, N \rangle$. The firmware state S is assumed to contain the following special elements:

1. A set of bit-vector variables $S.mem = \{S.memop, S.memaddr, S.datain, S.dataout\}$ which contain information about the current memory operation. $S.memop = \mathsf{NOP}$ means that the current instruction does not read/write from memory or memory-mapped I/O, $S.memop = \mathsf{RD}$ denotes a read and WR is a write. $S.memaddr$ is the address of the current memory operation, $S.datain$ is the data read *from* memory (or I/O), and $S.dataout$ is the data being written to memory or I/O. These variables help model memory-mapped I/O and accesses to untrusted memory.
2. $S.ROM$ contains the read-only memory which contains the firmware to be executed. We assume that the instruction and data memories are separate.
3. The set $S.regs$ which contains all the remaining state variables of the ILA. $S.regs$ contains the special element $S.PC$ which refers to the program counter of the microcontroller.

The initial state of the microcontroller, i.e., the state from which execution starts is defined by the symbolic expression $init_S$. N is the next state function, and N_{mem} and N_{regs} are the projections of N to the sets $S.mem$ and $S.regs$, respectively.

13.3.3.2 A Review of Symbolic Execution

Our verification algorithm builds on symbolic execution using constraint solvers [9, 21]. In this section, we provide a brief overview of these techniques. Section 13.3.4 will describe the extensions required to verify information flow properties.

```
1  void foo(int a, int b) {
2      int c;
3      if (a < 0 || b < 0)
4          c = 0;
5      else
6          c = a+b;
7
8      assert (c >= a && c >= b);
9      return c;
10 }
```

Listing 13.1 Example code to demonstrate symbolic execution

Consider the code shown in Listing 13.1. To make understanding the algorithm easier, the code is shown in a C-like language but symbolic execution is actually performed on the equivalent binary code. For simplicity, let us assume that the line numbers shown in the listing correspond to program counter (PC) values. The following steps outline how a typical symbolic execution algorithm verifies the correctness of the assertions on line 8.

Execution starts in the symbolic state $init_S \triangleq S.PC = 3$. This means the initial state constrains the value of $S.PC$ to be 3 but all other variables/registers/memory values are unconstrained, i.e., they can take arbitrary values. The algorithm uses a constraint solver to enumerate all concrete values of $S.PC$ consistent with this symbolic state. In this case, this is just $S.PC = 3$. The conjunction of $S.PC = 3$ and $init_S$ is pushed onto a stack. This stack contains all paths in the program and associated symbolic states that need to be explored by the algorithm.

Symbolic Execution Main Loop

The main loop of the symbolic execution algorithm starts by popping the symbolic state at the top of the stack. Note symbolic states in the stack always have a specific (concrete) value for $S.PC$, so we know what instruction is to be executed next. This instruction is retrieved from the program memory and is executed symbolically, which means that we create symbolic expressions corresponding to the state update for each register and memory value in the program.

In our example, initially $S.PC = 3$, this instruction is a conditional branch which updates the PC according to the following expression: $ite(a < 0 \vee b < 0, 4, 6)$. In the above formula, *ite* is the if-then-else operator and the formula states that the next PC value will be 4 if $a < 0$ or $b < 0$, and 6 otherwise. All other variables and memory states are left unchanged; they retain the same values as in $init_S$.

Now the algorithm uses a constraint solver to evaluate all values of $S.PC$ consistent with these next state expressions. This gives us $S.PC = 4$ and $S.PC = 6$. The conjunction of each of these, $init_S$, and a *path condition* is now pushed onto

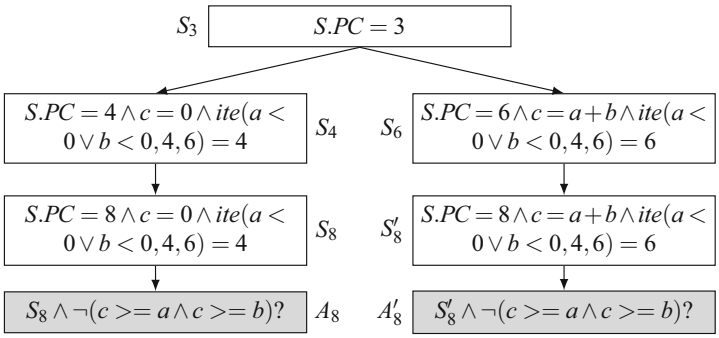


Fig. 13.8 State evolution in symbolic execution

the stack. The path condition records the constraints under which this branch is taken. Figure 13.8 shows that the path condition for symbolic state S_4 is $ite(a < 0 \vee b < 0,4,6) = 4$. This path condition can be syntactically simplified to $a < 0 \vee b < 0$ and records the fact that line 4 is only reached when $a < 0$ or $b < 0$. We now have two paths to explore, one with $PC = 4$ which is reachable when $a < 0 \vee b < 0$ and another with $PC = 6$ which is reachable under the path condition $\neg(a < 0 \vee b < 0)$. When we reach final instruction, in this case $PC = 9$, nothing is pushed onto the stack, signifying that no additional paths need to be explored.

This loop is repeated until the stack is emptied, i.e., all paths are visited.

Verifying Assertions

The symbolic state expressions constructed when verifying the code in Listing 13.1 are shown in Fig. 13.8. The white boxes show the symbolic state expressions and are labelled with their corresponding PC values. We see that $PC = 8$ is reached on both paths in the program with different path conditions.

The gray boxes show the assertions on line 8. Assertions are verified by negating the assertion condition, conjoining this with the state expressions and checking if the result is satisfiable in a constraint solver. In this example, A_8 is not satisfiable, so the assertion holds along this path. But the A'_8 is satisfiable because the result of $c = a + b$ can overflow. This violation is reported by the algorithm.

The assertions shown in the listing are predicates on program states. Unfortunately, such predicates cannot be used to reason about information flow [34] and so the algorithm described above cannot be used to verify information flow properties. An algorithm that can verify information flow properties is given in Sect. 13.3.4.

13.3.4 Verifying Information Flow Properties

Algorithm 2 shows how information flow properties can be verified using symbolic execution. It performs a depth-first search (DFS) of all reachable instructions and checks whether the information flow property specified by ($src, dst, srcpred, dstpred$) holds for all of them. The *stack* keeps track of paths to be visited and the *path constraints* P_A and P_B determine the conditions under which each path is taken.

Algorithm 2 Symbolic execution

Inputs: $init_S$, $\langle src, srcpred, dst, dstpred \rangle$

1: $stack.push(\langle init_S, init_S, \texttt{true}, \texttt{true} \rangle)$
2: **while** $\neg empty(stack)$ **do**
3: ▷ S_A and S_B represent the current firmware state.
4: $\langle S_A, S_B, P_A, P_B \rangle \leftarrow stack.pop()$
5:
6: $T \leftarrow P_A \wedge P_B \wedge [\![dstpred]\!]_{S_A} \wedge [\![dstpred]\!]_{S_B} \wedge [\![dst]\!]_{S_A} \neq [\![dst]\!]_{S_B}$
7: **if** $sat(T)$ **then** ▷ check properties
8: **display** violation
9: **end if**
10:
11: **if** $finished(S_A, S_B)$ **then** ▷ check for completion
12: **continue**
13: **end if**
14:
15: ▷ S'_A and S'_B is state after this instruction is executed.
16: $S'_A.mem \leftarrow N_{mem}(S_A)$
17: $S'_B.mem \leftarrow N_{mem}(S_B)$
18:
19: ▷ check for MMIO and handle it
20: **if** $isMMIO(S'_A)$ **then**
21: $\langle S'_A, S'_B \rangle \leftarrow execMMIO(S_A, S_B)$
22: **end if**
23: ▷ rewrite *datain* if *memaddr* matches the source
24: **if** $overlaps(S'_A, src)$ **then**
25: $S'_A.datain = \texttt{ite}(overlaps(S'_A.memaddr, src) \wedge srcpred(S_A), newVar(), S'_A.datain)$
26: $S'_B.datain = \texttt{ite}(overlaps(S'_B.memaddr, src) \wedge srcpred(S_B), newVar(), S'_B.datain)$
27: **end if**
28: ▷ compute next value of the registers
29: $\langle S'_A.regs, S'_B.regs \rangle \leftarrow \langle N_{regs}(S_A), N_{regs}(S_B) \rangle$
30:
31: ▷ find all concrete values $\texttt{PC}_\texttt{i}$ consistent with the symbolic value $[\![PC]\!]_{S_A}$ and add to stack.
32: **for all** $\texttt{PC}_\texttt{i} \models [\![\text{PC}]\!]_{S_A}$ **do**
33: $(S'_A.PC, S'_B.PC) = (\texttt{PC}_\texttt{i}, \texttt{PC}_\texttt{i})$
34: $P_A \leftarrow P_A \wedge S_A.PC = \texttt{PC}_\texttt{i}$
35: $P_B \leftarrow P_B \wedge S_B.PC = \texttt{PC}_\texttt{i}$
36: $stack.push(\langle S'_A, S'_B, P_A, P_B \rangle)$
37: **end for**
38: **end while**

```
1  uint8_t tbl[] = { 1, 1 };    // address of tbl = 0x100
2  uint8_t data = 3;            // &data=0x102
3  uint8_t IO_REG = 1;          // &IO_REG=0x200.
4
5  void foo(int r1) {
6      if (r1 < 0 || r1 >= N) return;
7      IO_REG = tbl[r1];
8  }
```

Listing 13.2 Integrity property example: src=r1@, dst=*dataout*@, srcpred=`true` and dstpred=*memaddr* = `0x200` ∧ *memop* = WR

There are two main enhancements over previous symbolic execution engines [4, 9, 10, 15]. The first is that the engine maintains *two* copies of the state in S_A and S_B. This is so that we can functionally test whether assigning different values to *src* results in different values at *dst*. This check is performed in line 7. The substitution of the source with "fresh" unconstrained variables is performed in lines 24–27. The other difference is the handling of MMIO instructions in lines 20–22 which are executed using simulation.

To understand Algorithm 2, let us consider its execution on the code shown in Listing 13.2. We show the algorithm in C-like pseudocode to make understanding easier but the analysis is done on binary code. The property here states that the untrusted value `r1` must not influence the value of `IO_REG`.

Suppose, due to a typo `N`=3 instead of the correct value 2. The symbolic state computed by the algorithm when it reaches the assignment to IO_REG would be:

$P_A = \neg(x_A < 0 \vee x_A \geq 3)$	$P_B = \neg(x_B < 0 \vee x_B \geq 3)$
$S_A.dataout = \mathsf{M}_A[\texttt{0x100} + x_A]$	$S_B.dataout = \mathsf{M}_B[\texttt{0x100} + x_B]$
$S_A.memaddr = \texttt{0x200}$	$S_B.memaddr = \texttt{0x200}$
$S_A.memop = \mathsf{WR}$	$S_B.memop = \mathsf{WR}$
$S_A.\mathsf{M} = S_B.\mathsf{M} = [\texttt{0x100} \mapsto 1, \texttt{0x101} \mapsto 1, \texttt{0x102} \mapsto 3, \ldots]$	

In the above, P_A and P_B are the two path conditions which specify the conditions that must hold for this path in the program to the taken. $S_A.\mathsf{M}$ and $S_B.\mathsf{M}$ represent the values of the RAM. The state variables *dataout*, *memaddr*, and *memop* refer to the elements of *S.mem* which contain information about the memory operation being executed by this statement in each "copy" of the execution. x_A and x_B are the new variables created to represent the untrusted value `r1`. At this point, when the solver evaluates whether $dataout_A \neq dataout_B$ is possible along with P_A, P_B and the predicates, it will find $x_A = 1, x_B = 2$. This means that if `r1` = $x_A = 1$, then value stored in `IO_REG` is different from the value stored in `IO_REG` if `r1` = $x_B = 2$. This means the property being violated. Once we fix the bug and set `N`=2, then $(P_A, P_B) = (x_A \geq 0 \wedge x_A < 2, x_B \geq 0 \wedge x_B < 2)$. Now it is not possible make $dataout_A \neq dataout_B$ while satisfying P_A and P_B, so the algorithm will not report in an error.

An alternative technique for verifying information flow properties is *dynamic taint analysis* (DTA) [3, 13, 14, 29, 42, 46]. DTA works by associating a taint bit with each variable/object and propagating the taint bit from inputs to the outputs of a computation. DTA is a dynamic analysis: it is implemented by analysis of execution traces in which the source of the information flow property is tainted and the analysis checks if this taint propagates to the destination. DTA cannot search over the space of all possible executions.

Listing 13.2 is an example where DTA fails. Typically taints are tracked separately for memory addresses and memory data [42]. Therefore, if the address for a memory read is tainted but the data pointed to by the address is not tainted, the result of the memory read is *not* tainted. Under such a policy, the bug would not be detected by DTA as the taint would not propagate from `r1` to `IO_REG`. If we change the policy and taint the result of memory read when the address is tainted, we will have overtainting. DTA will report a problem even when the bug is fixed and `N`=2.

Now suppose src is `data`, while dst and dstpred are the same as before. This property states that the secret value `data` must not influence untrusted register `IO_REG`. Clearly, a violation exists if `N`=3 and it will be detected by Algorithm 2.

To detect this issue, DTA needs an instruction sequence where `r1`=2 in order to expose the violation. In other words, DTA cannot detect a violation without a trace that "activates" the problem. The above examples demonstrate the advantages of our technique: exhaustive static analysis of all program paths and states, and improved precision over DTA.

MMIO and Symbolic Execution

Memory-mapped I/O (MMIO) poses unique challenges for symbolic execution, because a read/write from/to MMIO might have "side-effects," e.g., initiating a hardware state machine. Ideally, we would model all these symbolically. However, this is difficult in practice because constructing a symbolic model for the entire MMIO space will be very time-consuming.

Therefore, we use selective symbolic execution to model the MMIO side-effects. Simulation models of the SoC are usually available during SoC design and these model MMIO accesses. We use these to execute the "side-effects" of MMIO reads/writes. This means we have to convert the symbolic expressions into sets of concrete values, execute the simulator for each concrete value, and then resume symbolic execution with the simulation results. This process is shown in Fig. 13.9b, which is contrast to fully symbolic execution shown in Fig. 13.9a. This process of conversion from symbolic to concrete states and back is tractable for firmware because typical accesses to only made to a small number of hardware registers.

Fig. 13.9 Execution model. In (**b**) the *shaded boxes* show single-path (concrete) execution through the simulation model due to MMIO. (**a**) Fully symbolic execution. (**b**) Selective symbolic execution

13.3.5 Evaluation

We evaluated our approach by examining part of the firmware of an upcoming commercial phone/tablet SoC. The SoC consists of a number of IPs for various functions such as display, camera, and touch sensing. This evaluation examined a single component IP, called the PTIP which is involved in security sensitive "flows" such as secure boot. It contains a proprietary 32-bit microcontroller which executes the firmware. The firmware interacts with the other IPs in the SoC through hardware registers accessed using MMIO.

13.3.5.1 Methodology

We synthesized an instruction-level abstraction (ILA) of the PTIP microcontroller and then used the ILA to generate a symbolic execution engine. Z3 v4.3.2 was the constraint solver [16] used. This symbolic execution engine was integrated with a pre-existing simulator for this microcontroller to model MMIO reads and writes to other parts of the SoC.

13.3.5.2 Security Objectives

The PTIP firmware interacts with system software, device drivers, and other untrusted IPs. Since these entities, especially the system software and drivers, may be compromised by malware, these are all untrusted. We explored two main security objectives as part of the evaluation. First, the PTIP memory holds a sensitive cryptographic key called the *IPKEY*. We verify that these untrusted entities cannot access *IPKEY*. Second, we verify control-flow integrity of the PTIP firmware. Three representative information flow properties we formulated to capture these security requirements.

The total size of the PTIP firmware is approximately a few tens of thousands of static instructions. Due to limited time, this evaluation focused on a set of message handler functions which send and receive commands/messages from the (untrusted) system software, drivers, and other IPs. The size of these handler functions was approximately several hundred static instructions.

13.3.5.3 Summary of Verification Results

In terms of scalability, the symbolic execution engine could explore up to about half a million instructions within the assigned time limit of 30 min. This was sufficient for exploring all possible paths in 4 out of 6 handlers examined in the evaluation. Full exploration of paths could not be completed for the other two handlers. While these results are promising and show that some real-world firmware can be examined, further improvements in scalability are likely possible with more sophisticated analysis techniques.

The PTIP firmware had previously undergone simulation-based testing and manual code review. However, we were still able to identify a tricky security bug that could lead to *IPKEY* exposure. Symbolic analysis involving reasoning over all possible input values was essential in helping discover this bug.

13.4 Discussion and Related Work

This chapter described a methodology for SoC security verification that is based on the construction of instruction-level abstractions and the use of symbolic simulation to verify information flow properties. We now discuss the applicability of potential extensions to this methodology. We also discuss related work.

13.4.1 SoC Security Verification

The ILA is a complete formal specification of hardware behavior that precisely defines the hardware/software interface. A key feature of the ILA is that it is machine-readable. Section 13.3 showed how symbolic execution on ILAs of microprocessors could be used to verify information flow properties in SoC designs. However, ILA-based verification is not limited to symbolic execution. Symbolic execution is just one way of carrying out the proof obligations posed by SoC security requirements. Other verification techniques, such as model checking and abstract interpretation, can also be used to carry out these proof obligations.

In particular, software model checkers [6, 11] can be more scalable than symbolic simulators because they avoid the path explosion problem by *implicitly* enumerating paths. However, current software model checkers do not support information flow properties. Therefore, extending model checkers with richer property specification

schemes in order to verify information flow properties is an important area for further work. Using ILAs in other formal frameworks such as interactive theorem provers may also be beneficial.

13.4.2 Related Work

There is a rich body of literature studying synthesis and verification. We survey some of the most closely related work below.

13.4.2.1 Synthesizing Abstractions

Our work [33, 49] builds on recent progress in syntax-guided synthesis which is surveyed in [2]. Our synthesis algorithm is based on oracle-guided synthesis from [26]. Our contribution is the use of synthesis for constructing abstractions of SoC hardware. Also related is the work of Godefroid et al. [22] They synthesize a model for a subset of the x86 ALU instructions using I/O samples. In comparison, our contributions are strong guarantees about the correctness of the synthesized abstraction and general abstractions for SoC hardware, not just ALU outputs. For example, they do not consider issues like the mapping between opcodes and instructions, the source registers, the memory addressing modes, how the PC is updated, etc. Our model considers all these details.

Also related is the work of Udupa et al. [52] They synthesize distributed protocols using a partial template specification and input/output traces. Similar to our work, they verify the correctness of the synthesized protocol using model checking. This idea of synthesis synergistically combining synthesis with model checking is also used by Gascon et al. [20] who synthesize cryptographic protocols which satisfy certain specifications.

13.4.2.2 SoC Verification

Refinement relations, used in proving the abstraction and the implementation match, are from [28, 35]. One approach to compositional SoC verification is by Xie et al. [54, 55] They suggest manually constructing a "bridge" specification that along with a set of hardware properties can be used to verify software components that rely on these properties. Our methodology makes it easy to construct the equivalent of the bridge specifications. Most importantly, it ensures correctness of the abstraction.

Horn et al. [24] suggest symbolic execution on a software model that contains both firmware and software models of hardware components. This approach is complementary to ours because it can be used for early design stage verification, when an RTL model may not be available. However, once the RTL model is constructed, there is no easy way of ensuring that the software model and the RTL are in agreement. This is the critical challenge addressed by our work.

13.4.2.3 Symbolic Execution and Taint Analysis

The DART and KLEE projects are the precursors of subsequent work in symbolic execution [9, 21]. They combined modern constraint solvers and dynamic analysis to generate tests for software programs. Subsequent projects, such as FIE and S^2E, have applied symbolic execution to firmware and low-level software [10, 15]. The most important difference between these frameworks and our work is that they only verify safety properties, *not* confidentiality and integrity.

A large body of work also studies dynamic taint analysis (DTA) [3, 29, 42, 46]. DTA suffers from both false positives and false negatives due to the problems of under- and over-tainting. Our work [50] does not result in false positives. An overview of DTA and symbolic execution is presented in [42]. We show how symbolic execution can be used to verify information flow; this is missing from [42] which treats DTA and symbolic execution separately.

13.5 Conclusion

In this chapter, we described a principled methodology for security verification of SoCs. The first component of the methodology is the construction of Instruction-Level Abstractions (ILAs) of SoC hardware components. The ILA of a hardware component is an abstraction that treats commands sent from firmware to the component as the equivalent of "instructions" and models all firmware-visible state updates due to these instructions. We described how ILAs can be semi-automatically synthesized and verified to be correct abstractions of hardware components.

The second component of the methodology is using the ILA for the verification of system-level security properties. We introduced a property specification language that can express requirements like confidentiality and integrity and an algorithm based on symbolic execution to verify these properties. Experimentally, we found that both components—ILA construction and verification using symbolic execution—helped to find several bugs in an SoC built out of open source components and part of a commercial SoC design.

References

1. S.V. Adve, K. Gharachorloo, Shared memory consistency models: a tutorial. IEEE Comput. **29**(12), 66–76 (1996)
2. R. Alur, R. Bodik, G. Juniwal, M.M.K. Martin, M. Raghothaman, S.A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in *Formal Methods in Computer-Aided Design* (2013)
3. G.S. Babil, O. Mehani, R. Boreli, M.-A. Kaafar, On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices, in *Security and Cryptography* (2013)

4. O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M.R. Tuttle, V. Zimmer, Symbolic Execution for BIOS Security, in *Proceedings of the 9th USENIX Conference on Offensive Technologies* (2015)
5. Berkeley Logic Synthesis and Verification Group, ABC: a system for sequential synthesis and verification (2014). http://www.eecs.berkeley.edu/~alanmi/abc/
6. D. Beyer, T.A. Henzinger, R. Jhala, R. Majumdar, The software model checker blast. Int. J. Softw. Tools Technol. Transfer **9**(5–6), 505–525 (2007)
7. M. Bohr, The new era of scaling in an SoC world, in *IEEE International Solid-State Circuits Conference-Digest of Technical Papers* (IEEE, New York, 2009), pp. 23–28
8. A.R. Bradley, SAT-based model checking without unrolling, in *Verification, Model Checking, and Abstract Interpretation* (2011)
9. C. Cadar, D. Dunbar, D. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in *Operating Systems Design and Implementation* (2008)
10. V. Chipounov, V. Kuznetsov, G. Candea, S2E: a platform for in-vivo multi-path analysis of software systems, in *Architectural Support for Programming Languages and Operating Systems* (2011)
11. E. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in *Tools and Algorithms for the Construction and Analysis of Systems* (2004)
12. J. Cong, M.A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, G. Reinman, Accelerator-rich architectures: opportunities and progresses, in *Proceedings of the 51st Annual Design Automation Conference* (ACM, New York, 2014), pp. 1–6
13. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, P. Barham, Vigilante: end-to-end containment of internet worms, in *Symposium on Operating Systems Principles* (2005)
14. J.R. Crandall, F.T. Chong, Minos: control data attack prevention orthogonal to memory model, in *IEEE/ACM International Symposium on Microarchitecture* (2004)
15. D. Davidson, B. Moench, S. Jha, T. Ristenpart, FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution, in *USENIX Conference on Security* (2013)
16. L. De Moura, N. Bjørner. Z3: an efficient SMT solver, in *Tools and Algorithms for the Construction and Analysis of Systems* (2008)
17. R.H. Dennard, V. Rideout, E. Bassous, A. LeBlanc, Design of ion-implanted MOSFET's with very small physical dimensions. IEEE J. Solid State Circuits **9**(5), 256–268 (1974)
18. H. Esmaeilzadeh, E. Blem, R.S. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, in *Proceedings of the International Symposium on Computer Architecture* (IEEE, New York, 2011), pp. 365–376
19. Experimental artifacts and synthesis framework source code (2016). https://bitbucket.org/spramod/ila-synthesis
20. A. Gascón, A. Tiwari, A synthesized algorithm for interactive consistency, in *NASA Formal Methods* (2014)
21. P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in *Programming Language Design and Implementation* (2005)
22. P. Godefroid, A. Taly, Automated synthesis of symbolic instruction encodings from I/O samples, in *Programming Language Design and Implementation* (2012)
23. S. Heule, E. Schkufza, R. Sharma, A. Aiken, Stratified synthesis: automatically learning the x86-64 instruction set, in *Proceedings of Programming Language Design and Implementation* (2016)
24. A. Horn, M. Tautschnig, C. Val, L. Liang, T. Melham, J. Grundy, D. Kroening, Formal co-validation of low-level hardware/software interfaces, in *Formal Methods in Computer-Aided Design* (2013)
25. H. Hsing, http://opencores.org/project,tiny_aes (2014)
26. S. Jha, S. Gulwani, S.A. Seshia, A. Tiwari, Oracle-guided component-based program synthesis, in *International Conference on Software Engineering* (2010)
27. S. Jha, S.A. Seshia, A theory of formal synthesis via inductive learning, in *CoRR*, abs/1505.03953 (2015)

28. R. Jhala, K.L. McMillan, Microarchitecture verification by compositional model checking, in *Computer-Aided Verification* (2001)
29. M.G. Kang, S. McCamant, P. Poosankam, D. Song, DTA++: dynamic taint analysis with targeted control-flow propagation, in *Network and Distributed System Security Symposium* (2011)
30. S. Krstic, J. Yang, D.W. Palmer, R.B. Osborne, E. Talmor, Security of SoC firmware load protocols, in *Hardware-Oriented Security and Trust*, pp. 70–75 (2014)
31. R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, D.M. Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction, in *Proceedings of International Symposium on Microarchitecture* (IEEE, New York, 2003), pp. 81–92
32. R. Lysecky, T. Givargis, G. Stitt, A. Gordon-Ross, K. Miller, http://www.cs.ucr.edu/~dalton/i8051/i8051sim/ (2001)
33. S. Malik, P. Subramanyan, Invited: specification and modeling for systems-on-chip security verification, in *Proceedings of the Design Automation Conference*, DAC '16, New York, NY (ACM, New York, 2016), pp. 66:1–66:6
34. J. McLean, A general theory of composition for trace sets closed under selective interleaving functions, in *IEEE Computer Society Symposium on Research in Security and Privacy* (IEEE, New York, 1994), pp. 79–93
35. K.L. McMillan, Parameterized verification of the FLASH cache coherence protocol by compositional model checking, in *Correct Hardware Design and Verification Methods* (Springer, Berlin, 2001)
36. A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, P. Nalla, GLA: gate-level abstraction revisited, in *Design, Automation and Test in Europe* (2013)
37. A.C. Myers, JFlow: practical mostly-static information flow control, in *Principles of Programming Languages* (1999)
38. M.D. Nguyen, M. Wedler, D. Stoffel, W. Kunz, Formal hardware/software co-verification by interval property checking with abstraction, in *Design Automation Conference* (2011)
39. A. Sabelfeld, A. Myers, Language-based information-flow security. IEEE Sel. Areas Commun. **21**, 5–19 (2003)
40. A. Sabelfeld, D. Sands, Declassification: dimensions and principles, J. Comput. Secur. **17**(5), 517–548 (2009)
41. R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P.P. Pande, C. Grecu, A. Ivanov, System-on-chip: reuse and integration. Proc. IEEE **94**(6), 1050–1069 (2006)
42. E. Schwartz, T. Avgerinos, D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in *IEEE Security and Privacy* (2010)
43. S.A. Seshia, Combining induction, deduction, and structure for verification and synthesis. Proc. IEEE **103**(11), 2036–2051 (2015)
44. R. Sinha, P. Roop, S. Basu, The AMBA SOC platform, in *Correct-by-Construction Approaches for SoC Design* (Springer, New York, 2014)
45. A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, V. Saraswat. Combinatorial sketching for finite programs, in *Architectural Support for Programming Languages and Operating Systems* (2006)
46. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M.G. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena, BitBlaze: a new approach to computer security via binary analysis, in *Information Systems Security* (2008)
47. J. Strömbergson, https://github.com/secworks/sha1 (2014)
48. P. Subramanyan, D. Arora, Formal verification of taint-propagation security properties in a commercial SoC design, in *Design, Automation and Test in Europe* (2014)
49. P. Subramanyan, Y. Vizel, S. Ray, S. Malik, Template-based synthesis of instruction-level abstractions for SoC verification, in *Formal Methods in Computer-Aided Design* (2015)
50. P. Subramanyan, S. Malik, H. Khattri, A. Maiti, J. Fung, Verifying information flow properties of firmware using symbolic execution, In *Design Automation and Test in Europe* (2016)
51. S. Teran, J. Simsic, http://opencores.org/project,8051 (2013)

52. A. Udupa, A. Raghavan, J.V. Deshmukh, S. Mador-Haim, M.M. Martin, R. Alur, TRANSIT: specifying protocols with concolic snippets, in *Programming Language Design and Implementation* (2013)
53. C. Wolf, http://www.clifford.at/yosys/ (2015)
54. F. Xie, X. Song, H. Chung, N. Ranajoy, Translation-based co-verification, in *Formal Methods and Models for Co-Design* (2005)
55. F. Xie, G. Yang, X. Song, Component-based hardware/software co-verification for building trustworthy embedded systems. J. Syst. Softw. **80**(5), 643–654 (2007)

Chapter 14
Test Generation for Detection of Malicious Parametric Variations

Yuanwen Huang and Prabhat Mishra

14.1 Introduction

Third-party IPs are widely used in SoC design methodology. Some of these IPs may come from untrusted third-party vendors. It is crucial to ensure that an IP block is not vulnerable to input conditions that violate its non-functional (parametric) constraints, such as power, temperature, or performance. Power supply voltages, increased integration densities, and higher operating frequencies, among other factors, are producing devices that are more sensitive to power dissipation and reliability problems.

Figure 14.1 shows a scenario where an adversary can construct a specific test (input sequence) that can maximize the pear power or the peak temperature. We use the terms *power virus* and *temperature virus* to refer to the tests that can violate the peak power and peak temperature, respectively. Excessive power dissipation can lead to overheating, electromigration, and a reduced chip lifetime. Moreover, large instantaneous power consumption causes voltage drop and ground bounce, resulting in circuit delays and soft errors. Therefore, accurate power estimation during the design phase is crucial to avoid a time-consuming re-design process and in the worst-case an extremely costly tape-out failure. As a result, reliability analysis has steadily become a critical part of the design process of digital circuits.

Figure 14.2 shows different types of power and temperature viruses for different IPs. For gate-level or RTL-level IPs, a power virus is a set of test vectors that can create excessive instantaneous power consumption. For a processor IP, a power virus

Y. Huang (✉) • P. Mishra
Computer and Information Science and Engineering Department, University of Florida, Gainesville, FL, USA
e-mail: yuanwenhuang@ufl.edu; prabhat@ufl.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_14

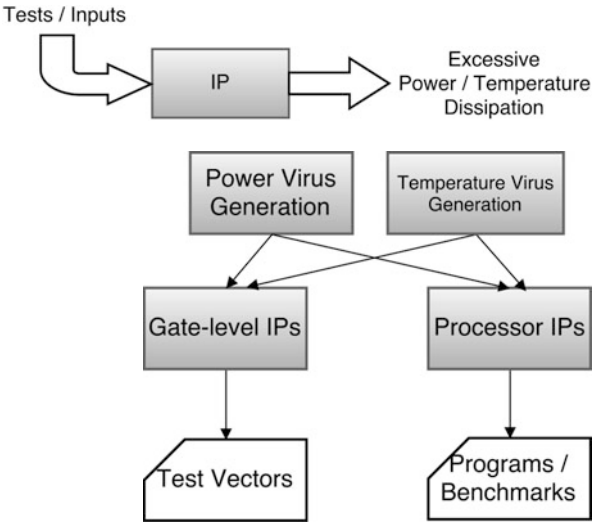


Fig. 14.1 Malicious inputs can cause excessive power/temperature dissipation

Fig. 14.2 Different types of power viruses

is a program that can cause peak power during its execution. Similarly, temperature virus generation can be for gate-level IPs or processor IPs. The reminder of this chapter describes these four test generation scenarios in detail.

14.2 Power Virus for Gate-Level IPs

The basic idea is to generate a power virus that can maximize the switching activity in the IP block. In case the peak power violates the threshold, the designers may need to re-design the IP block to reduce the peak power. In CMOS circuits, power dissipation depends on the extent of circuit switching activity, which is input-pattern dependent. The instantaneous power dissipation due to two consecutive input binary vectors is proportional to:

$$P \propto \sum_{\text{all gates}} T(g) * C(g) \tag{14.1}$$

where $C(g)$ denotes the output capacitance of gate g, and $T(g)$ indicates whether gate g switches or not when the circuit is fed with the two input vectors. $T(g)$ is 1 if the gate switches, and it is 0 if the gate does not switch. P is an estimation of the total power consumption.

The maximum circuit activity estimation problem aims at finding the input patterns which cause peak instantaneous dynamic power, a worst-case scenario where excessive simultaneous gate switching imposes extreme current demands on the power grid, leading to unwanted voltage drops. One naive approach is to exhaustively search for two consecutive binary input vectors which can induce the

maximum number of switching. Unfortunately, this problem is NP-complete for combinational as well as sequential circuits. The time complexity for this exhaustive search is $O(4^n)$, where n is the number of primary input pairs of the circuit and each input signal has four possible pairs of values (i.e., 0-0, 0-1, 1-1, 1-0).

Existing methods for maximum power estimation can be classified into two broad categories: simulation based and non-simulative approaches. For a circuit with large number of primary inputs, it is not possible to exhaustively search all input patterns for maximum power consumption. The only practical way to solve the problem is to generate a tight lower bound for maximum attainable power. During circuit simulation, Monte Carlo based technique can be used to estimate the maximum power [3, 22]. By estimating the mean and deviation of power and monitor the maximum power during simulation, a lower bound of the peak power can be measured from a statistical point of view. Non-simulative approaches use characteristics of the circuit and stochastic properties of input vectors to perform power estimation without explicit circuit simulation.

In the following subsections, we will explain in detail a few non-simulative methods for combinational circuits, including a pseudo-Boolean satisfiability approach and three other approaches based on Automatic Test Pattern Generation (ATPG). Then we will show how to extend the methods from combinational circuits to sequential circuits.

14.2.1 Pseudo-Boolean Satisfiability Approach

In CMOS circuits, the capacitance load of a gate output is proportional to the number of fanout. Hence, the power dissipation of two consecutive input vectors x^1 and x^2 can be rewritten as:

$$\sum_{i=1}^{m} F_i \cdot (g_i(x^1) \oplus g_i(x^2)) \tag{14.2}$$

where $g_i(*)$ denotes the Boolean function output of gate g_i with the specified input vector, and F_i denotes the fanout factor of gate g_i. This equation is an estimation of the total power consumption of the circuit, and it is the objective function to be maximized.

The pseudo-Boolean satisfiability approach was proposed in [14]. Figure 14.3b shows the new circuit N that is constructed for the pseudo-Boolean SAT problem. The new circuit N consists of two replicas of the original circuit T, namely T^1 and T^2. The primary input vectors (x^1 and x^2) are applied to the two replicas (T^1 and T^2), respectively. For every pair of corresponding gates, for example, g_1^1 in T^1 and g_1^2 in T^2, a new XOR gate xor_1 is constructed with g_1^1 and g_1^2 as inputs. The output of each XOR gate xor_i yields $g_i(x^1) \oplus g_i(x^2)$.

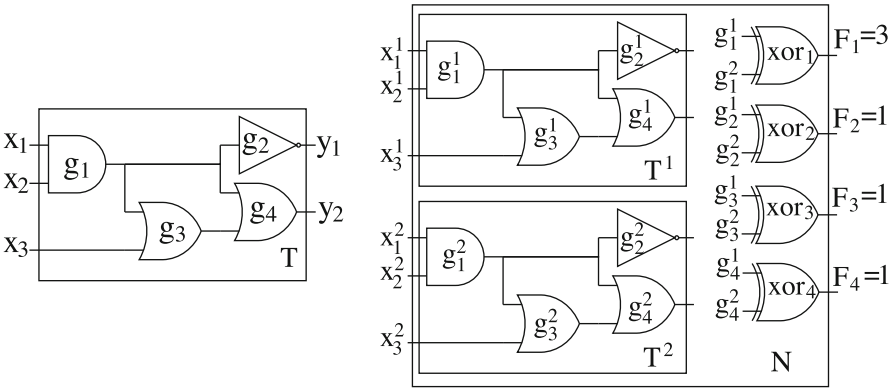


Fig. 14.3 Pseudo-Boolean optimization (PBO) formulation for combinational circuits [14]. (**a**) The original circuit. (**b**) The new circuit

The pseudo-Boolean SAT problem can be formed to maximize P as follows:

$$P = \sum_{i=1}^{m} F_i \cdot \text{xor}_i$$
$$\text{subject to} \quad \Psi = \text{CNF}(N) \tag{14.3}$$

The objective function P is the total number of switches in the original circuit. CNF(N) is the *Conjunctive Normal Form* of the new circuit N, which is the constraints that the pseudo-Boolean problem needs to satisfy.

The problem in Eq. (14.3) can be solved using pseudo-Boolean optimization (PBO) solvers. However, as a formal method, this approach is not scalable due to the complexity of pseudo-Boolean SAT problem for large circuits.

14.2.2 Largest Fanout First Approach

Instead of directly searching for input vector pairs (x^1, x^2) to maximize P, the Largest Fanout First approach [22] assigns transitions to the internal gates in a greedy way as shown in Fig. 14.4. The gates are sorted by the fanout number in decreasing order. In every iteration, a gate g_i which is not tried and has the largest fanout is selected to assign a transition: $g_i(x^1) \oplus g_i(x^2) = 1$. The assignment of gate g_i is justified by two processes: *backtracking* (backward to inputs) and *implication* (forward to outputs) in the circuit. If the justification of the transition assignment fails, i.e., conflicts happen during justification, the node values of the circuit will be

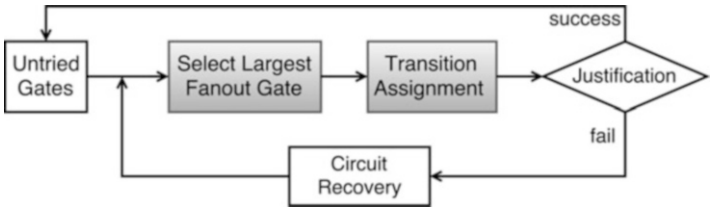


Fig. 14.4 Largest fanout first approach [22]

recovered, which means the newly implied values will be reset to previous values. The algorithm tries gates one by one to assign transitions until all gates have been processed. Interested readers can refer to [22] for details of the justification process.

14.2.3 Cost-Benefit Analysis Approach

The Largest Fanout First (LFF) approach [22] is a greedy algorithm that assigns transitions to high-fanout gates first. However, LFF neglects an important fact that when a specific gate (node) is marked for switching, it does create certain number of switching events but it may restrict some other switching in the future. To address the flaw in LFF , the cost-benefit approach [7] suggests that every assignment should have a cost-benefit analysis. After applying a certain assignment to a gate, there might be changes in future switching possibility for all other untried gates. The cost-benefit analysis on all possible assignments selects the most favorable assignment, which facilitates the overall optimization process to enable more efficient power virus generation. The *switching probability* of a gate and the *cost-benefit* function for an assignment are defined as follows:

$$\begin{aligned} \mathrm{SP}(g_i) &= \frac{\text{number of fanin combinations that make } g_i \text{ switch}}{\text{total number of available fanin combinations}} \\ \mathrm{CB}(a) &= \sum_{\text{untried gates}} \Delta\mathrm{SP}(g_i) * F_i \end{aligned} \tag{14.4}$$

where a is the assignment made at the current iteration and F_i is the fanout number of gate g_i. $\mathrm{SP}(g_i)$ is the switching probability of gate g_i, which is the ratio of the number of fanin combinations that can make g_i switch to the total number of available fanin combinations. $\Delta\mathrm{SP}(g_i)$ is the difference in switching probability after and before the assignment. If $\Delta\mathrm{SP}(g_i)$ is positive, this assignment increases the switching probability of gate g_i. In this case, $\Delta\mathrm{SP}(g_i) * F_i$ is considered as the benefit because of this assignment; otherwise, it is considered as the cost. $\mathrm{CB}(a)$ is the sum of benefit/cost of all untried gates. The assignment with the largest $\mathrm{CB}(a)$ is the most beneficial one, and it is most favorable for untried gates to switching in the future.

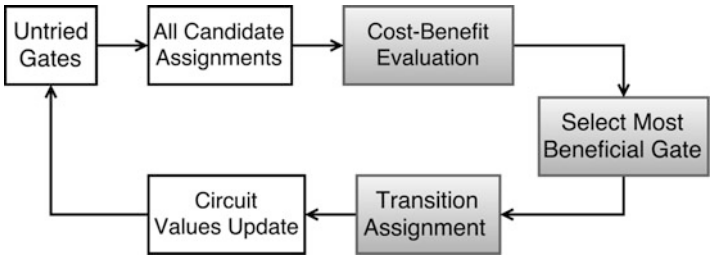


Fig. 14.5 Cost-benefit analysis approach [7]

The workflow for the cost-benefit approach is shown in Fig. 14.5. At the beginning of each iteration, the cost-benefit analysis is done for all possible assignments of all untried gates. The cost-benefit analysis also involves justification of the assignment throughout the circuit. Only feasible assignments are evaluated for cost-benefit values. The gate with the most beneficial assignment is selected, and the assignment is applied to the circuit to update values in the circuit. The algorithm stops when all gates have been tried. Interested readers can refer to [7] for details.

14.2.4 Power Virus for Sequential Circuits

So far, we have described how to construct power virus for combinational circuits. The estimation of power consumption for sequential circuits [9, 10, 16, 17] is a natural extension of the above approaches. The dynamic power of the combinational portion in a sequential circuit is still the same as shown in Eq. (14.1). The problem of single-cycle peak power in a sequential circuit is very similar to that of instantaneous power in combinational circuit. The only difference is that sequential circuits have state elements (flip-flops or storage elements). Let S_i denote the state of flip-flops at the beginning of the *i-th* clock cycle. Let I_i denote the primary input vector for the *i-th* clock cycle. The sequence $(S_1, I_1, S_2, I_2, \ldots, I_n, S_{n+1})$ represents the change of states of the circuit with a set of input vectors.

The approaches discussed above for power virus generation for combinational circuits can be extended to sequential circuits for single-cycle peak power generation. The outputs of one clock cycle depend on both the input vector and the state of flip-flops from previous cycle. To apply the above approaches, we need to initialize the state of flip-flops to a known state. If the circuit is full-scan design, the state of the flip-flops can be initialized to any arbitrary value. For circuits without a full scan chain inserted, the initialization can be done by *warming up* the circuit as discussed in [10, 17]. The initial state of flip-flops are assumed to be undefined and input vectors are generated to feed the circuit until all the flip-flops hold a definite value. In the case that the initial state of a circuit is not fully controllable, the above procedure cannot initialize some of the flip-flops. As stated in [10], random values

can be speculated to these flip-flops. Once the state of the circuit is defined, we can apply the approaches for power virus generation for combination circuits, which is to find a pair of input vectors for maximum power consumption. In Sect. 14.4, we will discuss more about *peak k-cycle power* and *peak sustainable power*.

14.3 Power Virus for Processor IPs

A power virus in an architecture-level IP, or a processor, is a computer program that can execute specific machine code to have excessive CPU power consumption. Since the cooling system of a computer is designed with the budget of thermal design power, rather than the maximum power, a power virus could cause the system to overheat. If the power management logic cannot stop the processor in time, the power virus may permanently damage the system. In order to design a suitable power budget and various power management features, it is crucial to understand the power characteristics of the system and get accurate estimation of the attainable worst-case power dissipation. It is important to note that power virus for processor is a special case of power virus discussed above.

14.3.1 Stress Benchmarks

Power virus programs, or stress benchmarks, are often used for thermal testing of computer components during the design phase. Thermal testing is part of the stability testing when designing and benchmarking a reliable computer system. MPrime [15] and CPUburn-in [2] are the most popular benchmarks to stress-test a CPU.

MPrime [15] is a freeware application that searches for Mersenne prime numbers. It is called the torture test which has been extremely popular among PC enthusiasts and overclockers as a stability testing utility. The stress-test feature can be configured by setting the size for fast Fourier transform during its primality check. The program can subject the processor and memory to an incredibly intense workload, and it halts when it encounters even one minor error. The amount of time that a processor remains successfully stable while running MPrime is used as a measure of the system's stability. This feature has made MPrime suitable to test the stability of processor and memory, as well as the cooling efficiencies.

CPUburn-in [2] is a stability testing tool for overclockers, written by Michal Mienik. The program heats up any x86 processor to the maximum possible operating temperature. The program constantly runs FPU intensive functions for a user specified period of time, and it continuously monitors for errors ensuring that the CPU does not generate errors under overclocking conditions.

The torture tests such as MPrime [15] and CPUburn-in [2] can put heavy workload on the processor and push the system towards high power dissipation.

But there is no guarantee that the worst-case power dissipation can be attained by these benchmarks. Chip designers have to write hand-crafted power virus programs for a good estimation of the attainable peak power. However, it is very tedious and inefficient to manually design effective power virus for a given architecture. Moreover, power virus designed for one specific architecture usually cannot be directly used for a different processor IP. An automatic exploration and code generation methodology is needed to generate power viruses for architecture-level IPs. The following two subsections will introduce such an automatic approach which can generate power viruses for any architecture using genetic algorithm.

14.3.2 Power Virus Generation for Single-Core IPs

The goal is to generate a power virus program that can create the maximum worst-case power consumption. It is important to note that the worst-case power of a processor is not simply the sum of the maximum power of all components. It is almost impossible to have all components/resources achieve the maximum utilization at the same time. Due to the stalls in processor pipelines and contention of other shared resources, such as cache or memory ports, the peak total power is significantly smaller than the sum of peak power of all components.

When an instruction goes through different pipeline stages (fetch, decode, execute, memory, and commit), it will activate certain functional units and components in the processor. The power consumed by one instruction depends on three factors: (1) the opcodes and operands of the instruction, (2) the other instructions that are competing resources with this instruction, (3) the hardware constraints of the architecture. A power virus consists of a sequence of instructions that would create maximum power in a given architecture.

It is crucial that if the power virus can evolve and adapt to the hardware configurations of the processor IP, so as to best utilize all the components at all stages of the pipeline. Machine learning and genetic algorithms can be used to drive this process to search for power viruses with maximum power consumption. Such a learning approach (such as [12] and [5]) is not constrained by the hardware configurations, thus is applicable to different processor IPs.

The framework for automatic generation of synthetic power virus programs is shown in Fig. 14.6. The *Genetic Algorithm* generates the parameter values for the potential candidates for the power viruses as it searches through the parameter space. These abstract program parameters are fed to the *Code Generator* that generates a synthetic C program containing embedded assembly instructions based on these specified parameters. The C programs will be compiled into binary programs, which are candidates for power viruses. The simulator executes each binary program and gets the estimated maximum power consumption. The power consumption for each binary program will be the fitness value, which indicates the fitness of the set of program parameters used to generate this program. The *Genetic Algorithm* will take all the fitness values as feedback, tune the parameters to intelligently search for next

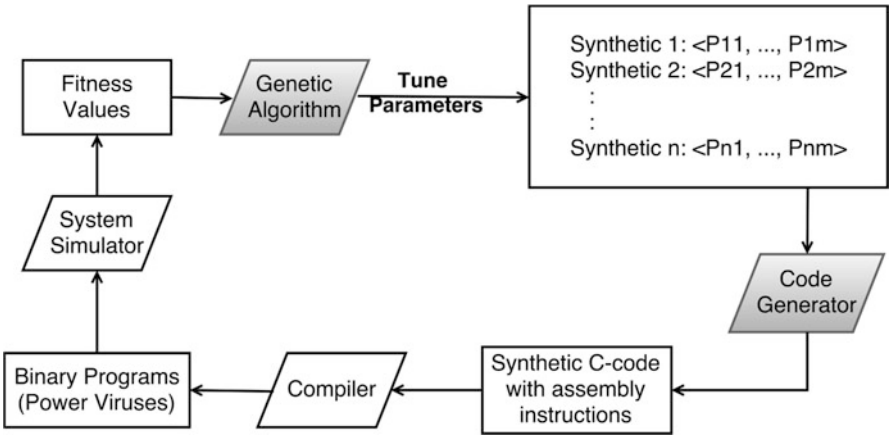


Fig. 14.6 A framework to construct synthetic benchmarks (power virus) using genetic algorithm [5]

Table 14.1 Search space and parameter settings [5]

Category	Program parameter	Range
Control flow predictability	Number of basic blocks	(10–200)
	Average block size	(10–100)
	Average branch predictability	(0.8–1.0)
Instruction mix weights	Integer instructions (ALU, mul, div)	(0–4, 0–4, 0–4)
	FP instructions (add, mul, div)	(0–4, 0–4, 0–4)
	Load/store instructions (ld, st)	(0–4, 0–4)
Instruction level parallelism	Register dependency distance distribution	
Data locality	Stride value distribution for load/store	
	Data footprint	(1–200)
Memory level parallelism	Mean of memory level parallelism	(1–6)

generation of parameters which can have even higher power consumption (fitness value). The process iteratively continues until the genetic algorithm converges to find the best power virus for a given system configuration. The program parameter space is presented in Table 14.1 in Sect. 14.3.2.1. The process of *Code Generator* is explained in detail in Sect. 14.3.2.2.

14.3.2.1 Exploration Space of Program Characteristics

Given a specific architecture, we would like to adaptively change the parameters of the power virus program to maximize power consumption. The parameters of the program include control flow predictability, instruction mix parameters, instruction

level parallelism, data locality, and memory level parallelism. The exploration space of all these parameters is shown in Table 14.1.

The control flow predictability of the program is set by three different parameters: the number of basic blocks, the average block size, and the average branch predictability. The number of basic blocks and block size defines the instruction footprint of the program. The branch predictability of a program is important as it affects the overall throughput of the pipeline. When a misprediction happens, the pipeline has to be flushed and this would result in reduced activity in the pipeline. The percentage of correctly predicted branches dictates the activity level of the branch predictor. The power consumption of a branch predictor is usually around 5 % of the overall processor power.

The instruction mix determines the frequency of each type of instruction in the program. Let us consider eight types of instructions (three types of integer instructions, three types for FP instructions and load/store instructions) in Table 14.1. Since different instructions have various latencies and power consumption, the instruction mix has a major impact on the overall power consumption. The typical power consumption of the integer and floating point ALU is around 4–6 % and 6–12 %, respectively.

The instruction level parallelism is determined by the register dependency distance distribution. The register distance between instructions can be (1, 2, 4, 8, 14, 16, 20, 32, 48, 64) and the program will search for the best distribution in these ten bins. This parameter impacts the throughput of the pipeline directly. The overall throughput of the pipeline is related to the power consumption of the instruction window, clock, and rename logic. The typical power consumption for the instruction window and the clock subsystem is around 8 % and 20 %, respectively.

The data level parallelism is determined by data locality and memory level parallelism. Data locality includes two aspects: the stride value between load/store instructions, the data footprint (the number of iterations before resetting to the beginning of the data array). The stride value between the addresses of load/store instructions can be (0, 4, 8, 12, 16, 32, 64) and the program will search for the best distribution. Data footprint controls the number of cache lines that will be touched by load/store instructions. The average number of long-latency outstanding loads (loads with big strides that will cause miss in the last level cache) represents the memory level parallelism. Interested readers can refer to [5] for details.

14.3.2.2 Code Generation

The process of code generation is illustrated in Fig. 14.7. First, the number of basic blocks in the synthetic program is fixed. For each basic block, the instruction type for every instruction using the instruction mix is selected. The basic blocks are then bound together using conditional jumps. For each instruction, one needs to find a producer instruction to assign a register dependency. The destination registers are assigned by round-robin. The source registers are assigned based on dependency. Memory access is modelled by having load/store accesses to a set of 1-D arrays

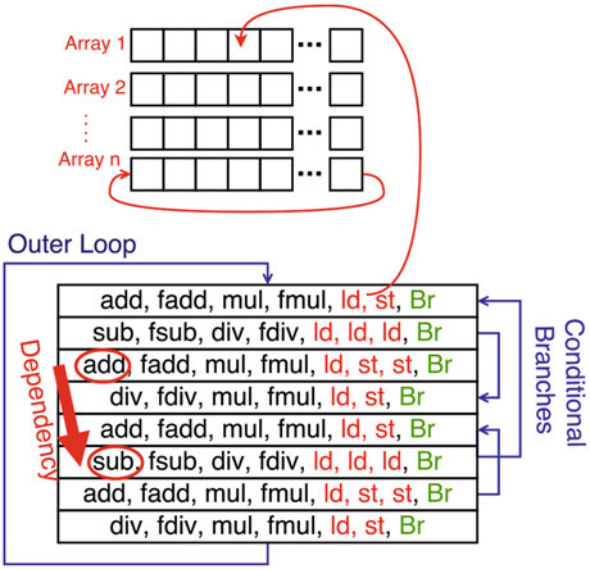


Fig. 14.7 Example of code generation of a synthetic power virus program [5]

in a stride fashion. Load/store instructions are grouped into pools and assigned to different arrays. The pointer of arrays are reset to the beginning of array when required data footprint is touched. Interested readers can refer to [5] for details.

14.3.3 Power Virus for Multi-Core IPs

For a multi-core IP, a power virus would be a multi-threaded program that has maximum power consumption. It is more challenging than single-core IP because of components like the interconnection network, shard caches, DRAM, and coherence directory, which also contribute significantly to the power consumption of a multi-core parallel system. We need to exploit these features of multi-core systems to the right extent. The parameter exploration space should take these features into consideration and use genetic algorithm to search for optimal parameters [1, 4, 13].

14.3.3.1 Space Exploration of Program Characteristics

Table 14.2 shows the program parameters for multi-threaded power virus generation. The first two categories (parallelism and shared data access) are specifically for multi-thread exploration, and the rest of the parameters are the same as in Table 14.1. The number of threads controls the amount of thread level parallelism of the synthetic program. The thread class and process assignment parameter choose various patterns which decides the assignments of threads to cores that they are

Table 14.2 Search space and parameter settings for multi-core IP [4]

Category	Program parameter	Range
Parallelism	Number of threads	(1–32)
Shared data access	Thread class and processor assignment	
	Percent memory accesses to shared data	(10–90)
	Shared memory access strides	(0–64)
	Coupled load-stores	(True/false)
Control flow predictability	Number of basic blocks	(10–200)
	Average block size	(10–100)
	Average branch predictability	(0.8–1.0)
Instruction mix weights	Integer instructions (ALU, mul, div)	(0–4, 0–4, 0–4)
	FP instructions (add, mul, div)	(0–4, 0–4, 0–4)
	Load/store instructions (ld, st)	(0–4, 0–4)
Instruction level parallelism	Register dependency distance distribution	
Data locality	Stride value distribution for load/store	
	Data footprint	(1–200)
Memory level parallelism	Mean of memory level parallelism	(1–6)

bound to execute. The percentage of memory accesses to shared data and the shared memory access strides will influence accesses to shared data. A coupled load-store is a load with a following store, which mimics a migratory sharing pattern of access. The coupled load-store parameter can be either set or unset.

14.3.3.2 Multi-Threaded Power Virus Generation

Figure 14.8 shows the flowchart of automatic generation of multi-threaded power viruses using genetic algorithm. The work flow is very similar to the power virus generation for single core, except that there are more parameters related to the multi-thread characteristics. Each set of parameters contain specification for the multi-thread characteristics as well as for each thread. Interested readers can refer to [4] for details.

14.4 Temperature Virus

14.4.1 Temperature Virus for Gate-Level IPs

Temperature virus can be built on the idea of power virus, because sustained high power could produce peak temperature. The sequence $(S_1, I_1, S_2, I_2, \ldots, I_n, S_{n+1})$ represents the change of states of the circuit with a set of input vectors. Figure 14.9 shows the definitions of *peak single-cycle power*, *peak n-cycle power*, and *peak*

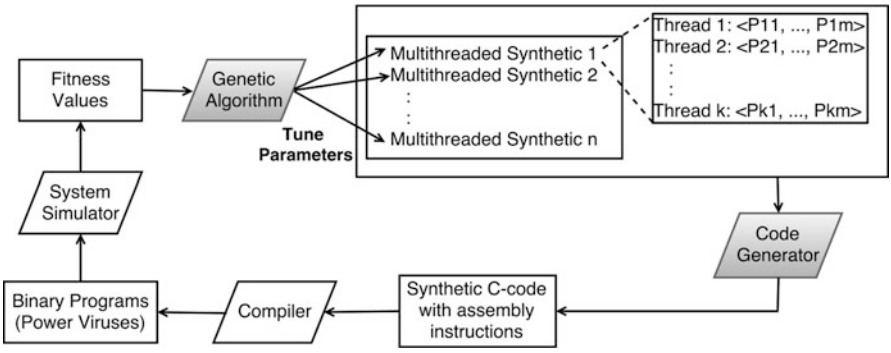


Fig. 14.8 A framework for multi-threaded power virus generation using genetic algorithm [4]

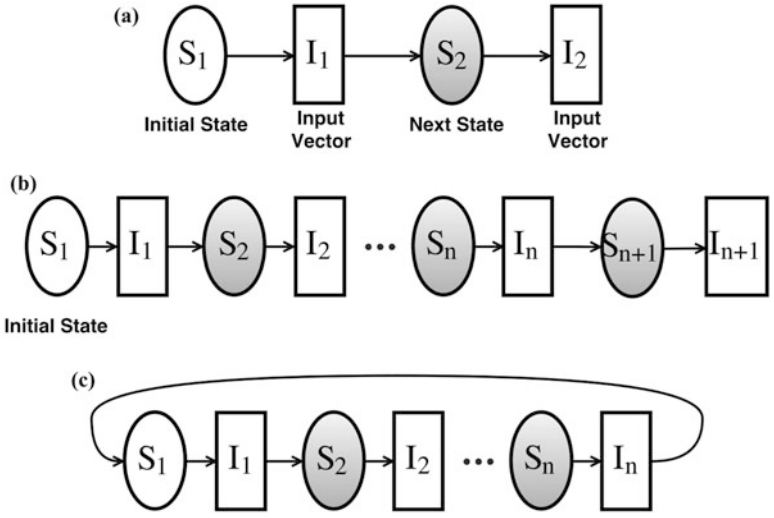


Fig. 14.9 Definitions of peak power measures. (**a**) Peak single-cycle power. (**b**) Peak n-cycle power (power measured and averaged over n cycles). (**c**) Peak sustainable power (power measured and averaged over n cycles, circuit returns initial state for every n cycles) [8]

sustainable power. For *peak single-cycle power*, the power virus would be a tuple (S_1, I_1, I_2) that has the maximum power consumption during one clock cycle. For *peak n-cycle power*, the power virus would be a sequence $(S_1, I_1, I_2, \ldots, I_n, I_{n+1})$ that has the maximum average power consumption over n clock cycles. For *peak sustainable power*, the power virus can make the circuit return to the initial state S_1 after n clock cycle. The power virus which can produce the *peak sustainable power* is actually a temperature virus, because we can repeatedly apply the input sequence $(I_1, I_2, \ldots, I_n)$ to have the circuit indefinitely remain on high power consumption.

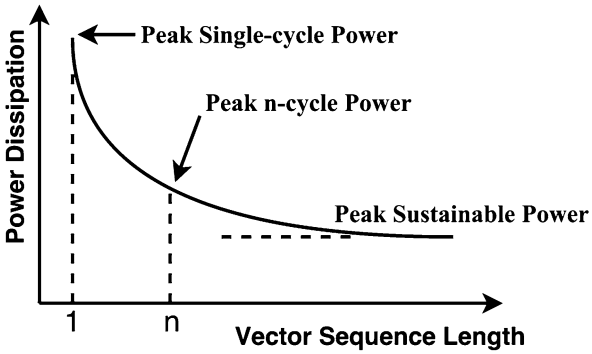


Fig. 14.10 Lower bound for peak power dissipation [8]

The initial state S_1 is important in determining the peak power/temperature of the sequential circuit. If the circuit state is fully controllable, the approaches for power virus generation discussed in Sect. 14.2 can be extended for *peak single-cycle power* and *peak n-cycle power* estimation. Genetic algorithm can also be used to generate the sequence $(S_1, I_1, I_2, \ldots, I_n, I_{n+1})$, with the initial state S_1 chosen from a pool of reachable and controllable states.

For *peak sustainable power* (peak temperature), the problem is to derive a long sequence of vectors that have high power dissipation which can be sustained. Hsiao et al. [8] have shown a typical trend for *peak n-cycle power* dissipation and the process of searching for *peak sustainable power*. As shown in Fig. 14.10, when n equals to 1, it is the *peak single-cycle power*. As n increases, the *peak n-cycle power* is expected to decrease if the vector sequence cannot sustain the power dissipation. The peak power levels off when n is large enough or a loop is found in the state transitions. There are two approaches to search for power virus with peak sustainable power. The first one is to start with a *peak n-cycle power* sequence, and then search for vectors to close the loop with as few state transitions as possible. But the initial state might take a lot of cycles to reach. Moreover, the state transitions to close the loop can act as a cool-down stage, which might reduce the average power dissipation. The second approach is to start with an initial state which is easy to reach, and return to the easy state with very few additional transitions. Hsiao et al. [8] used the second approach and pushed it to an extreme that they start from entirely *don't care* states. It takes one extra cycle to return to the initial state, since *don't care* states is superset of all states. The sequence generated by [8] would be one that has transitions from any unknown state to another state, which can be repeatedly applied to the circuit indefinitely.

14.4.2 Temperature Virus for Processor IPs

The approaches discussed in Sect. 14.3 for power virus generation for processor IPs can be directly applied to temperature virus generation. The genetic algorithm based approach can search for the synthetic program which has the best fitness with the

specified system configurations. We can simply change the fitness function from peak power to sustainable power or peak temperature. The new fitness function will direct the genetic algorithm towards the search for temperature viruses.

14.5 Conclusion

In this chapter, we discussed power/temperature viruses that might cause excessive heating of IPs. At the gate level, we illustrated test generation techniques to generate automatic power viruses for both combinational and sequential circuits. At the system level, genetic algorithm can be used to automatically and effectively generate synthetic assembly programs for a given processor architecture. The test vectors and synthetic programs of power viruses can be very useful in maximum power estimation and thermal threshold selection. The power viruses can be used to test the power/thermal behaviors of a chip and the robustness of the power/thermal management design. In general, modern processor IPs have very advanced dynamic power management (DPM) and dynamic thermal management (DTM) features, such as dynamic voltage/frequency scaling [20, 21] and dynamic reconfiguration [6, 11, 18, 19, 23–25]. A power virus needs to bypass all those DPM/DTM techniques to cause real harm to the system.

References

1. R. Bertran, A. Buyuktosunoglu, M.S. Gupta, M. González, P. Bose, Systematic energy characterization of cmp/smt processor systems via automated micro-benchmarks, in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, December (2012), pp. 199–211
2. CPUburn-in. http://cpuburnin.com/
3. N.E. Evmorfopoulos , G.I. Stamoulis, J.N. Avaritsiotis. A Monte Carlo approach for maximum power estimation based on extreme value theory. IEEE Trans. Comput. Aided Des. **21**(4), 415–432 (2002).
4. K. Ganesan, L.K. John, MAximum Multicore POwer (MAMPO): an automatic multithreaded synthetic power virus generation framework for multicore systems, in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, Article No. 53 (2011)
5. K. Ganesan, J. Jo, W.L. Bircher, D. Kaseridis, Z. Yu, L.K. John, System-level Max Power (SYMPO) - a systematic approach for escalating system-level power consumption using synthetic benchmarks. in *The 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010
6. H. Hajimiri, P. Mishra, S. Bhunia, Dynamic cache tuning for efficient memory based computing in multicore architectures, in *International Conference on VLSI Design*, 2013
7. H. Hajimiri, K. Rahmani, P. Mishra, Efficient peak power estimation using probabilistic cost-benefit analysis, in *International Conference on VLSI Design*, Bengaluru, January 3–7, 2015
8. M.S. Hsiao, E.M. Rudnick, J.H. Patel, K2: an estimator for peak sustainable power of VLSI circuits, in *IEEE International Symposium on Low Power Electronics and Design*, 1997

9. M.S. Hsiao, E.M. Rudnick, J.H. Patel, Effects of delay models on peak power estimation of VLSI sequential circuits, in *Proceedings of the 1997 IEEE/ACM International Conference on Computer-aided Design*, 1997
10. M.S. Hsiao, E.M. Rudnick, J.H. Patel, Peak power estimation of VLSI circuits: new peak power measures. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **8**(4), 435–439 (2000)
11. Y. Huang, P. Mishra, Reliability and energy-aware cache reconfiguration for embedded systems, in *IEEE International Symposium on Quality Electronic Design*, 2016
12. A.M. Joshi, L. Eeckhout, L.K. John, C. Isen, Automated microprocessor stressmark generation, in *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)* (2008), pp. 229–239 (2008)
13. Y. Kim, L.K. John, S. Pant, S. Manne, M. Schulte, W.L. Bircher, M.S.S. Govindan, AUDIT: stress testing the automatic way, in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* December (2012), pp. 212–223
14. H. Mangassarian, A. Veneris, F. Najm, Maximum circuit activity estimation using pseudo-Boolean satisfiability. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **31**(2), 271–284 (2012)
15. MPrime, wikipedia page. https://en.wikipedia.org/wiki/Prime95
16. K. Najeeb, V.V.R. Konda, S.K.S. Hari, V. Kamakoti, V.M. Vedula, Power virus generation using behavioral models of circuits, in *25th IEEE VLSI Test Symposium (VTS'07)*, Berkeley, CA (2007), pp. 35–42
17. K. Najeeb, K. Gururaj, V. Kamakoti, V. Vedula, Controllability-driven power virus generation for digital circuits, in *IEEE 20th International Conference on VLSI Design (VLSID)* (2007), pp. 407–412
18. X. Qin, W. Wang, P. Mishra, TCEC: temperature- and energy-constrained scheduling in real-time multitasking systems. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. (TCAD) **31**(8), 1159–1168 (2012)
19. K. Rahmani, P. Mishra, S. Bhunia, Memory-based computing for performance and energy improvement in multicore architectures, in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2012
20. W. Wang, P. Mishra, Pre-DVS: preemptive dynamic voltage scaling for real-time systems with approximation scheme, in *ACM/IEEE Design Automation Conference (DAC)* (2010), pp. 705–710
21. W. Wang, P. Mishra, System-wide leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in multitasking systems. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. (TVLSI) **20**(5), 902–910 (2012)
22. C.Y. Wang, K. Roy, Maximum power estimation for CMOS circuits using deterministic and statistical approaches. IEEE Trans. VLSI **6**(1), 134–140 (1998)
23. W. Wang, P. Mishra, S. Ranka, Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems, in *ACM/IEEE Design Automation Conference (DAC)* (2011), pp. 948–953
24. W. Wang, P. Mishra, S. Ranka, *Dynamic Reconfiguration in Real-Time Systems - Energy, Performance, Reliability and Thermal Perspectives* (Springer, New York, 2013). ISBN: 978-1-4614-0277-0
25. W. Wang, P. Mishra, A. Ross, Dynamic cache reconfiguration for soft real-time systems. ACM Trans. Embed. Comput. Syst. (TECS) **11**(2), article 28, 31 pp. (2012)

Part V
Conclusion

Chapter 15
The Future of Trustworthy SoC Design

Prabhat Mishra, Swarup Bhunia, and Mark Tehranipoor

15.1 Summary

This book provided a comprehensive coverage of IP security and trust issues with contributions from academic researchers, SOC designers as well as SoC verification experts. The topics covered in this book can be broadly divided into the following three categories.

15.1.1 Trust Vulnerability Analysis

Chapter 1 highlighted how security of IP can be compromised at various stages in the overall SoC design-fabrication-deployment cycle affecting various parties. This book presented five efficient techniques for trust vulnerability analysis.

- *Security Rule Check:* Chap. 2 presented a framework to analyze design vulnerabilities at different abstraction levels and assessed its security at design stage.
- *Vulnerability Analysis at Gate and Layout Levels:* Chap. 3 described vulnerability analysis for both gate- and layout-level designs to quantitatively determine their susceptibility to hardware Trojan insertion.
- *Code Coverage Analysis:* Chap. 4 presented an interesting case study to identify suspicious signals using both formal and semi-formal coverage analysis methods.
- *Layout Analysis for Probing Attack:* Chap. 5 surveyed existing techniques in performing probing attacks, protection against probing attacks, and presented a layout-driven framework to assess designs for vulnerabilities to probing attacks.

P. Mishra (✉) • S. Bhunia • M. Tehranipoor
University of Florida, Gainesville, FL, USA
e-mail: prabhat@ufl.edu; swarup@ece.ufl.edu; tehranipoor@ece.ufl.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_15

- *Side Channel Vulnerability:* Chap. 6 covered side channel testing using three metrics as well as practical case studies on real unprotected and protected targets.

15.1.2 Effective Countermeasures

The next two chapters provided effective countermeasures against various attacks. The goal of these approaches is to make it hard for the attacker to introduce vulnerability.

- *Camouflaging, Encryption, and Obfuscation:* Chap. 7 reviewed three major security hardening approaches—camouflaging, logic encryption/locking, and design obfuscation—that are applied to ICs at layout, gate, and register transfer levels.
- *Mutating Runtime Architecture:* Chap. 8 presented a mutating runtime architecture to support system designers in implementing cryptographic devices hardened against side channel attacks.

15.1.3 Security and Trust Validation

The final six chapters presented efficient techniques for validation of IP security and trust vulnerabilities.

- *Validation of IP Trust:* Chap. 9 surveyed the existing security validation methods for soft IP cores using a combination of simulation-based validation and formal methods.
- *Proof-Carrying Hardware:* Chap. 10 utilized model checking and theorem proving using proof-carrying hardware for trust evaluation.
- *Trust Verification:* Chap. 11 described three methods to detect potential hardware Trojans by utilizing Trojan characteristics. It also outlined how a stealthy Trojan can evade these methods.
- *Unspecified IP Functionality:* Chap. 12 outlined how to exploit unspecified functionality for information leakage and presented a framework for preventing such attacks.
- *Security Property Validation:* Chap. 13 proposed mechanism for specifying security properties and verifying these properties across firmware and hardware using instruction-level abstraction.
- *Malicious Parametric Variations:* Chap. 14 described how to perform test generation for detecting malicious variations in parametric constraints.

These chapters provided a comprehensive coverage of hardware IP trust analysis as well as effective trust validation techniques to enable secure and trustworthy SoC design.

15.2 Future Directions

Although significant research has been carried out over the past decade for securing IPs, there are still many challenges ahead, especially as the IC and IP supply continuously change. For example, the IP vendors are shifting towards using encryption to protecting their IPs from piracy. This would make logic simulation, trust verification, and integration with other IPs in an SoC more difficult. Furthermore, addressing one security issue may create unwanted new vulnerabilities to some other security concerns. We briefly outline some of the challenges ahead in verifying security and trustworthiness of future IPs.

15.2.1 Security and Trust Verification for Encrypted IPs

Recent trends in IP piracy have raised serious concerns to IP developers. IP piracy can take several forms, for example, an untrusted SoC designer may legally purchase a third party IP (3PIP) core from an IP vendor and then make illegitimate copies of the original IP and sell them under their own name [4]. The SoC designer may also add some extra features to the original IPs to make them look like a new one and then sell them to another SoC designer for making easy profit. To prevent IP piracy, the IP developers are increasingly adopting the IEEE P1735 encryption scheme developed by Design Automation Standards Committee of the IEEE [7]. Most EDA tools also support IEEE P1735 standard which utilizes two levels of encryption to produce an encrypted IP core [8]. IEEE P1735 standard ensures that the IP in plaintext format is never exposed to the SoC integrator while allowing the EDA tools to perform functional simulation, synthesis, etc., of the encrypted IP core.

Unfortunately, IEEE P1735 encryption scheme introduces new challenges and complications for IP trust verification and security rule check by restricting the SoC designer to analyze the RTL code. This would prohibit the SoC integrator from applying some of the IP trust verification techniques proposed in this book as well as in the literature. For example, IP trust verification techniques proposed in [1–3, 5, 6, 12] cannot be applied on encrypted IPs as they require analysis of RTL code which needs to be in plaintext format. The academic research community should focus their effort to ensure that IP trust verification is possible even when the IP is given in an encrypted format. One possible solution is to investigate IP trust verification techniques which work on the gate-level netlist. The reason is that most IP providers allow the visibility of the gate-level netlist in unencrypted format according to IEEE P1735 standard.

15.2.2 Security and Trust Verification for Obfuscated IPs

Logic obfuscation approach does not address all the issues associated with IP piracy. The untrusted SoC integrator can add some additional functionality to the obfuscated IP and sell it to other SoC integrators as a firm IP. For example, an untrusted SoC integrator may purchase an encryption engine from an IP provider in encrypted format. Then, the SoC integrator can include hashing functionality with the encryption engine and synthesize them to gate-level netlist (unencrypted format). The untrusted SoC integrator can then illegitimately sell it as a firm IP which can perform encryption and hashing operation to other SoC designers under its own name. To address this issue, academic researchers have proposed to obfuscate and functionally lock the IP [10, 11]. This approach works by placing locking gates (XOR/ XNOR) into the design. The IP produces functionally correct output when it receives the correct chip unlock key. The obfuscation approaches have gained interest in the industry and are expected to be included in design flows in the near future.

The IP obfuscation technique would help address the problem of IP piracy. However, from SoC designer's point of view it would make the IP trust verification extremely challenging. Similarly, logic obfuscation makes it difficult to perform security rule check. To date, no IP trust verification approach has been proposed in the literature to take IP obfuscation into account. Most existing techniques like FANCI [16], rare net identification [13], security rule check [18] for IP trust verification would not work on netlist which is functionally locked. The academic research community should direct their attention and effort to developing new and innovative IP trust verification techniques that can be applied to obfuscated and functionally locked IPs.

15.2.3 Security and Trust Verification for Hard IPs

Another growing concern in the industry is verifying the trust of hard IPs. Most foundries have begun to offer more and more hard IPs to SoC designers. These hard IPs have comparatively lower cost, have a high yield since they have been produced many times before and been tested for high yield, and offer minimal time to market delay. However, these IPs are incorporated in the design at the very last stages of SoC design flow (physical layout). Any analysis done on the layout level would be very complex and time consuming. One possible solution to this problem would be to take inspiration from postsilicon debugging techniques and develop postsilicon security and trust verification techniques.

15.2.4 Security and Trust Verification During SoC Design Flow

It is of utmost importance to identify IP security and trust issues at all levels of abstraction in the SoC design flow and during transition from one stage to the next stage of the design process. In most literature, the 3PIP vendor and the foundry are considered as untrusted [17]. However, there are other entities in the SoC design flow that can maliciously incorporate hardware Trojans in the design or create security vulnerability. For example, many SoC designers outsource the design for test (DFT) insertion task to third party entities. These entities can incorporate malicious circuitry in the design before returning it to the SoC designer. Therefore, it is important for the SoC developers to not only verify trustworthiness of third party IP but also to verify trustworthiness of their design at subsequent levels of abstraction, namely synthesis, DFT insertion, physical layout, etc. Researchers can draw inspiration from different fields in order to address this problem. For example, techniques proposed for IP piracy prevention can be adopted in order to establish trust in the design. The SoC designers can obfuscate and functionally lock their design and restrict other entities in the SoC design flow from making any malicious modifications.

15.2.5 Unintentional Vulnerabilities

Many security vulnerabilities in SoCs could be created unintentionally by CAD tools and by designers' mistakes. CAD tools are extensively used for synthesis, DFT insertion, automatic place and route, etc. However, today's CAD tools are not equipped with understanding security vulnerabilities in SoCs. Therefore, the tools can introduce additional vulnerabilities in the design. For example, during synthesis process, the tool tries to optimize a design for power, area, and/or performance of the design. If there exists any don't-care conditions in the RTL specification, the synthesis tool introduces deterministic values for the don't-care conditions for design optimization. This could facilitate attacks such as fault injection or side channel-based attacks [18]. Another example worthy of mentioning here would be the vulnerabilities introduced by the DFT tool. The inserted DFT can create numerous vulnerabilities by allowing attackers to control or observe internal assets of an SoC. Vulnerabilities in an SoC design can also be introduced by designer's mistakes.

Traditionally, the design objectives are driven by cost, performance, and time-to-market constraints, while security is generally neglected during the design phase. In [9], authors have shown that if a finite state machine is designed without security into consideration, it can introduce vulnerabilities by facilitating fault injection attack. It is of paramount importance to identify these security vulnerabilities during hardware design and validation process. However, given the growing complexity

of modern SoC designs, it is extremely difficult, if not impossible, to manually identify these vulnerabilities. This calls for the development of CAD tools which can automatically evaluate the security of a design in reasonable time without significantly impacting time-to-market. Also to avoid some common security problems in early design stages, security-aware design practices must be developed and used as guidelines for design engineers.

15.2.6 Multi-Security Objectives Design

The evaluation time for IP security and trust verification and finding vulnerabilities in an IP need to be scalable with the design size in order to have low impact on time-to-market. It is therefore important to identify which security analysis methods and countermeasures need to be applied given the application of the target SoC. For example, for a crypto IP, side channel vulnerability analysis and mitigation are recommended. However, for non-crypto IP the side channel leakage may not pose any threat and therefore, side channel vulnerability analysis and mitigation are not required for such IPs. However, if an SoC designer blindly starts applying side channel leakage countermeasures for all IPs, it would not only cause unnecessary area overhead but also could have negative impact on Trojan detection approaches which rely on side channel evaluation. In summary, addressing one security issue may negatively impact another security feature, hence during design process multi-security objectives must be considered. To perform this analysis, a designer needs to select the security vulnerabilities he/she may be concerned about for the target SoC, and the tool should perform an optimization process to ensure highest security for all vulnerabilities.

15.2.7 Metrics and Benchmarks

Finally, a major limitation for the performance evaluation of any IP security and trust verification technique is the inadequate number and quality of benchmark circuits. Benchmarks and metrics are necessary components for establishing baseline for comparison between different techniques developed by the research community. The Trojan benchmark circuits developed in [14, 15] are not adequate to help address some of the challenges raised by the emerging IP trust issues, such as encrypted IPs. There is currently no encrypted IP with Trojan in [15]. Therefore, it is important for the research community to design new and innovative metrics and Trojan benchmark circuits that incorporate these features.

References

1. F. Farahmandi, Y. Huang, P. Mishra, Trojan localization using symbolic algebra, in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2017
2. X. Guo, R. Dutta, Y. Jin, F. Farahmandi, P. Mishra, Pre-silicon security verification and validation: a formal perspective, in *ACM/IEEE Design Automation Conference (DAC)*, 2015
3. X. Guo, R. Dutta, P. Mishra, Y. Jin, Scalable SoC trust verification using integrated theorem proving and model checking, in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2016), pp. 124–129
4. U. Guin, Q. Shi, D. Forte, M.M. Tehranipoor, FORTIS: a comprehensive solution for establishing forward trust for protecting IPs and ICs. ACM Trans. Des. Autom. Electron. Syst. (TODAES) **21**(4), 63 (2016)
5. Y. Huang, S. Bhunia, P. Mishra, MERS: statistical test generation for side-channel analysis based Trojan detection, in *ACM Conference on Computer and Communications Security (CCS)*, 2016
6. M. Hicks, M. Finnicum, S.T. King, M.M.K. Martin, J.M. Smith, Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *Proceedings of IEEE Symposium on Security and Privacy* (2010), pp. 159–172
7. IEEE Approved Draft Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP) (2014)
8. Microsemi 2014, *Libero SoC Secure IP Flow User Guide for IP Vendors and Libero SoC Users* (2014). http://www.microsemi.com/document-portal/docview/133573-libero-soc-secure-ip-flow-user-guide
9. A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, M. Tehranipoor, AVFSM: a framework for identifying and mitigating vulnerabilities in FSMs, in *Design Automation Conference*, 2016
10. M.T. Rahman, D. Forte, Q. Shi, G.K. Contreras, M. Tehranipoor, CSST: preventing distribution of unlicensed and rejected ICs by untrusted foundry and assembly, in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (2014), pp. 46–51
11. J.A. Roy, F. Koushanfar, I.L. Markov, EPIC: ending piracy of integrated circuits, in *Proceedings of the on Design, Automation and Test in Europe* (2008), pp. 1069–1074
12. H. Salmani, M. Tehranipoor, Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level, in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (2013), pp. 190–195
13. H. Salmani, R. Karri, M. Tehranipoor, On design vulnerability analysis and trust benchmarks development, in *Proceedings of IEEE 31st International Conference on Computer Design (ICCD)*, pp. 471–474 (2013)
14. H. Salmani, M. Tehranipoor, R. Karri, On design vulnerability analysis and trust benchmark development, in *IEEE International Conference on Computer Design (ICCD)*, 2013
15. Trust-HUB, http://trust-hub.org/resources/benchmarks
16. A. Waksman, M. Suozzo, S. Sethumadhavan, FANCI: identification of stealthy malicious logic using Boolean functional analysis, in *Proceedings of the ACM Conference on Computer and Communications Security* (2013), pp. 697–708
17. K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, M. Tehranipoor, Hardware Trojans: lessons learned after one decade of research. ACM Trans. Des. Autom. Electron. Syst. **22**(1), article 6 (2016)
18. K. Xiao, A. Nahiyan, M. Tehranipoor, Security rule checking in IC design. Computer **49**(8), 54–61 (2016)

Index

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0