FORMAL VERIFICATION OF HARDWARE SECURITY AND TRUST

By

FARIMAH FARAHMANDI

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2018

ACKNOWLEDGMENTS

First of all, I really appreciate my adviser Prof. Prabhat Mishra for what he has done for me. It was him who led me to open the door of computer science. I will never forget all his kind instructions and enduring support. He not only guided me to overcome challenging problems, but also taught me how to explore new directions. More importantly, he is always considerate to me and has helped me building my own career. He is the person who made this dissertation come true.

I would also like to thank my other Ph.D. committee members: Prof. Sartaj Sahni, Prof. Sanjay Ranka, Prof. Nima Maghari for their precious advises and criticisms. I appreciate Prof. Sandip Ray with whom I collaborated for one year. His profound knowledge in post-silicon validation and debug helped me a lot in my research. I am also thankful for the collaboration with Prof. Domenic Forte and Prof. Mark Tehranipoor.

I also thank my lab-mates, Yuanwen Huang, Alif Ahmed, , Yangdi Lyu, Subodha Charles, and Jonathan Cruz. It was my great pleasure to work with them. I really enjoyed our friendship and I hope it will last forever.

Last but not least, I sincerely thank my mother, father, and my lovely sisters (Farzaneh and Fargol) for their love, encouragement and support. I wouldn't be able to achieve anything without my parents' raising since my childhood. Their all-embracing love and care – guiding me patiently, supporting my decisions, forgiving my faults – makes me grow up freely. I want to thank my beautiful friend, Roshanak, who has been always by my side and help me a lot. My most special appreciation is dedicated to my Husband, Matin, his love and devotion paved the road to my doctoral degree. He always holds my hands and gives me courage whenever I need. I have realized how lucky I am to have him.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

FORMAL VERIFICATION OF HARDWARE SECURITY AND TRUST

By

Farimah Farahmandi

April 2018

Chair: Prabhat Mishra
Major: Computer Engineering

Trust establishment in semiconductor designs has become a major challenge for design houses and government since several countries and companies are involved during different stages of a design life cycle. Hardware Trojans are malfunctions which can be inserted during any design stage such as defining specification, implementing designs at different abstraction levels, layout extraction and manufacturing. A triggered hardware Trojan can severely affect the integrity and security of the circuit by causing system failures such as granting an unauthorized access to secret information. Hardware Trojans are designed in a way that they are inactive most of the time and can be triggered with a very rare input sequence. Therefore, conventional validation techniques are not effective to detect them because of the Trojan's stealthy nature. Formal methods are promising to prove the security properties; however, the conventional formal methods suffer from scalability concerns. In this dissertation, I propose scalable hardware trust validation techniques using formal methods. I have developed an automated test generation and debugging framework using Gröbner basis reduction for arithmetic circuits. For pre-silicon designs, I have shown that symbolic algebra can be effectively used for localizing hardware Trojans. I have also developed efficient post-silicon trust validation techniques by leveraging pre-silicon verification effort. The experimental results demonstrate that the proposed approaches are scalable in establishing that the implementation has "nothing more, nothing less" compared to the golden specification.

CHAPTER 1
INTRODUCTION

People have embraced a wide variety of mobile devices in their daily lives in addition to their traditional desktop computers and laptops. Considering the fabric of Internet of Things (IoT) [115], where the number of connected devices exceeds the human population, we should all agree to the fact that computing devices pervade every aspect of our lives. In IoT devices, integrated electronics, sensors, sophisticated software and firmware, and learning algorithms are employed to make physical objects smart and adjustable to their environment. These highly complex and smart IoT devices are embedded everywhere - starting from household items (e.g., refrigerators, slow cookers, ceiling fans), wearable devices (e.g., fitness trackers, smart glasses, air bugs), medical devices (e.g., insulin pump, asthma monitoring, ventilator) to cars. These IoT devices are connected to each other as well as cloud in order to provide a real-time aid on a daily basis. Given the diverse and critical applications of these computing devices, it is crucial to verify the correctness, security and reliability of these devices.

Modern computing devices are designed using System-on-Chip (SoC) technology. In other words, SoC is the backbone for most of the IoT devices. Sheer complexity of the System-on-Chip (SoC) designs used in these systems combined with diverse and evolving use cases make product security assurance a major challenge. Effective and well-developed security validation is an important and integral part of the security assurance process today. However, existing security verification approaches both pre- and post-silicon are often ad-hoc and manual (i.e. rely on human ingenuity and experience). There is a critical need to identify all possible security vulnerabilities and fix them using an automatic and reliable mechanism during security validation. Attacks on hardware can be more effective and efficient than traditional software attacks since patching is extremely difficult (almost impossible) on hardware designs. Therefore, a security attack can be successfully repeated on every instance of a vulnerable IoT device. In other words, hardware level vulnerabilities are extremely important to be fixed before deployment since it affects the overall system security. Based on Common Vulnerability Exposure (CVE-MITRE) estimates, if hardware-level vulnerabilities are removed, the overall system vulnerability will reduce by 43% [67, 68].

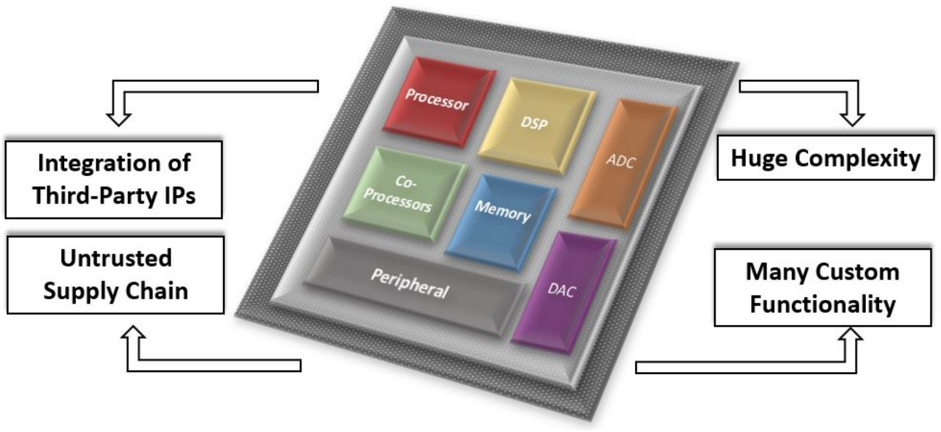


Figure 1-1. An SoC design integrates a wide variety of IPs in a chip. It can include one or more processor cores, on-chip memory, digital signal processor (DSP), analog-to-digital (ADC), and digital-to-analog converters (DAC), controllers, input/output peripherals, and communication fabric. Huge complexity, many custom designs, distributed supply chain and integration of untrusted third-party IPs make post-silicon validation challenging.

An SoC architecture typically consists of several pre-designed Intellectual Property (IP) blocks, where each IP implements a specific functionality of the overall design. Figure 1-1 shows a typical SoC with its associated IPs. These IPs communicate with each other through Network-on-Chip (NoC) or standard communication fabrics. The IP-based design approach is popular today since it enable a low-cost design while meeting stringent time-to-market requirements. With the globalization of the IC industry, the outsourcing and integration of third-party hardware IPs (Intellectual Property) has become a common practice for System-on-Chip (SoC) design [17, 51]. However, it raises major security concerns as the attacker can insert malicious modifications in third-party IPs and tamper the system. Security vulnerabilities can be inserted in high-level specification (e.g., TLM and RTL models) in synthesized gate-level netlist, layouts as well as in the fabricated chip by an attacker. It is extremely important to differentiate trustworthy designs from untrustworthy ones by detecting any potential malicious functionality (widely known as hardware Trojans), which is not in the design specification. Hardware Trojans consist of two main parts: trigger and payload. The trigger is a condition which activates the Trojan circuit and the payload is the part of the circuit (functionality) which can be affected by an activated Trojan. A triggered Trojan may endanger the integrity of the design by leaking critical information such as secret keys or causing denial of service.

A major challenge for Trojan identification is that Trojans are usually stealthy [17]. 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. Therefore, it is difficult to activate a Trojan and even more difficult to detect or locate it. As a result, conventional validation methods are impractical to detect hardware Trojans. Conventional structural and functional testing methods are not effective to activate trigger conditions since there are many possible Trojans and it is not feasible to construct a fault model for each of them. As a result, existing EDA tools are incapable of detecting hardware Trojans to differentiate between trustworthy third-party IPs and untrusted ones.

There has been a lot of research on hardware Trojan detection using logic testing and side channel analysis [2, 30, 76]. Logic testing focuses on generating efficient tests to activate a Trojan and check the primary output values of specification and circuit-under-test to detect Trojan. On the other hand, side channel analysis compares signatures of specific physical characteristics (e.g., dynamic current) between the design and the golden specification. However, none of these methods can effectively establish the trust for a given design. Therefore, it is crucial to have methods that can detect, localize and eliminate the Trojan, and produce Trojan-free circuits. Formal methods are suitable for detecting and localizing Trojans during the design phase. In this dissertation, I focus on different formal methods to verify the design and detect undesired functions (possibly

Trojans). As shown in Figure 2-1, I proposed trust validation based on Formal methods that includes efficient test generation using property (model) checking, satisfiability problem, equivalence checking and reaching effective coverage closures in post-silicon.

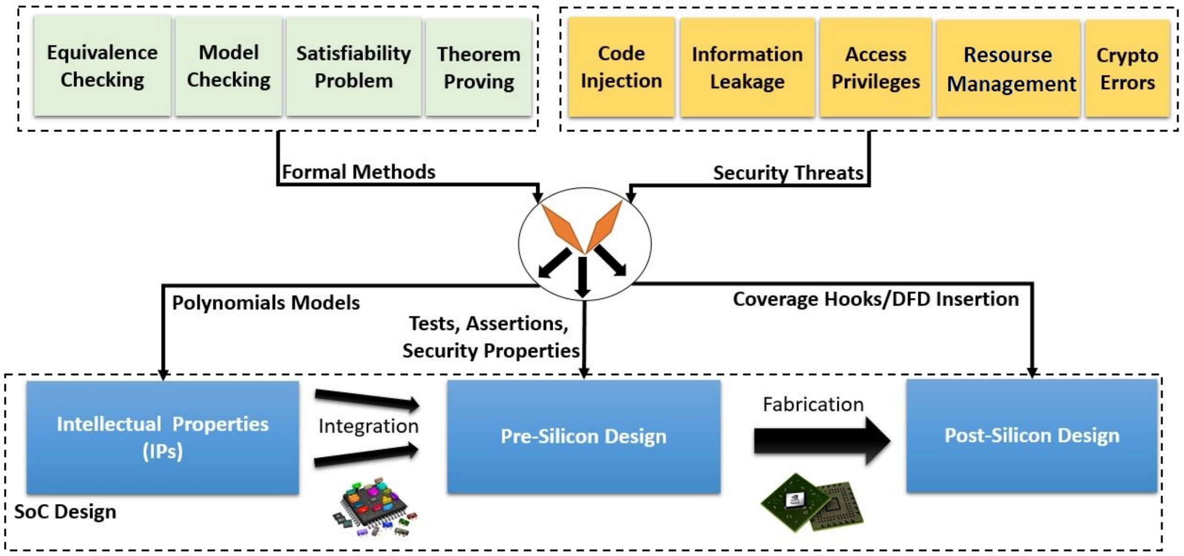


Figure 1-2. Hardware trust verification can be categorized in three major directions: i) high-level modeling using polynomials, ii) generation of tests, assertions and properties for post-silicon security validation, and iii) effective utilization of pre-silicon efforts for post-silicon debug.

The rest of this chapter is organized as follows. Section 1.1 describes the security vulnerabilities in current hardware design flow as well as associated challenges to validate them. Section 1.2 summarizes the contributions of this dissertation. Finally, Section 1.3 describes the organization of this dissertation.

1.1 Security Validation: Opportunities and Challenges

From specification to silicon, the design will go through different transformations. System on chip design procedure starts with defining the specification. The specification contains all information about the expected behavior of the chip. In the second step, the specification is implemented using programming languages such as Verilog and VHDL. Then, the modules should be synthesized to a gate-level netlist. The gate-level netlist will go through several non-functional changes for different purposes such as clock-tree insertion, power optimization. The layout of the gate-level netlist is extracted, and the chip will be fabricated in the final design step. During each of these steps, there may be several security concerns that threaten the integrity of the whole chip. In other words, it is extremely important to make sure that the design is behaving like the specification in each of the steps and each intermediate design forms should be exactly equal to the specification, nothing more, nothing less.

A design can encounter security threats during different stages of its life cycle as shown in Figure 1-3. Security threats can be inserted throughout the IC design as well as manufacturing process. In the pre-silicon stage, vulnerability can be introduced due to (1) designer mistakes, rogue employees, and untrusted third-party IPs during the design integration phase; (2) untrusted EDA tools in the synthesize phase; (3) untrusted EDA tools and untrusted vendors when design for test (DFT), design for debug (DFD), and dynamic power management (DPM) functions are added. In the post-silicon stage, vulnerabilities can come from (1) untrusted foundry during manufacturing and (2) physical attacks or side channel attacks after the chip is shipped. It is of paramount importance to verify the trustworthiness of hardware IPs. Therefore, security validation is the critical part of the design process of digital circuits. In order to trust a design, verification and debugging should be done at each of the stages. In this section, we introduce some of the security validation challenges in both pre- and post-silicon phases.

1.1.1 Pre-Silicon Trust Validation

Any intentional or unintentional design mistakes (bugs or hardware Trojans) may cause security vulnerability in the design. Therefore, a design should be validated thoroughly. However, increasing complexity of integrated circuits increases the probability of Trojans in designs. To make it worse, the reduction in time-to-market puts a lot of pressure on verification and debug engineers to potentially faulty sign-off. The situation gets further exacerbated for arithmetic circuits as the bit blasting is a serious limitation for most of the existing validation approaches. Faster bug localization is one of the most important steps in design validation.

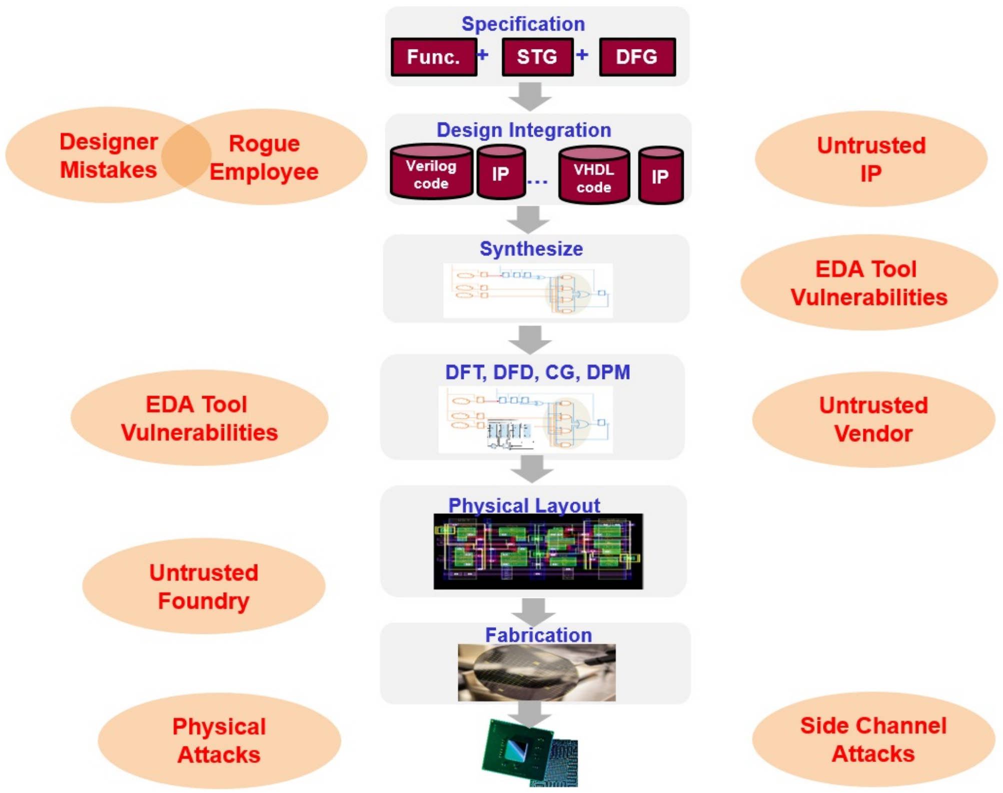


Figure 1-3. Potential threats during SoC design flow.

The urge of high speed and high precision computations increases use of arithmetic circuits in real-time applications such as cryptography operations (to ensure secure computation). Optimized and custom arithmetic architectures are required to meet the high speed and precision constraints. There is a critical need for efficient arithmetic circuit verification and debugging techniques due to error proneness of non-standard arithmetic circuit implementations. Hence, the automated debugging of arithmetic circuits is absolutely necessary for efficient security validation.

A major problem with design validation is that we do not know whether a Trojan exists, and how to quickly detect and fix it. We can always keep on generating random tests, in the hope of activating the bug; however, random test generation is neither scalable nor efficient when designs are large and complex. Existing directed test generation techniques [32, 33] are promising only when the list of faults (bugs) is available. However, they are not applicable when bugs are not known a prior.

Test generation is extremely important for functional validation of integrated circuits. A good set of tests can facilitate the debugging and help the verification engineer to find the source of problems. Test generation techniques can be classified into three different categories: random, constrained-random [3] and directed [32]. Random test generators are used to activate unknown errors; however, random test generation is inefficient when designs are large and complex. Constrained-random test generation tries to guide random test generator towards finding test vectors that may activate a set of important functional scenarios. The probabilistic nature of these constraints may lead to situations which can result in generating inefficient tests. Moreover, constraint generation is not possible when we do not have knowledge about the potential bug. A directed test generator, on the other hand, generates one test to target a specific functional scenario [32, 89]. Clearly, less effort is needed to reach the same coverage goal using directed tests compared to random or constrained-random tests. However, existing directed test generation methods require a fault list or desired functional behaviors that need to be activated [89]. These approaches cannot generate directed tests when the Trojan (faulty scenario) is unknown.

When effective tests are available, the source of Trojan has to be localized. Most of the traditional debugging tools are based on techniques such as simulation, binary decision diagrams (like BDDs,*BMD [25]) and SAT solvers [7, 97]. However, all of these approaches suffer from state space explosion while dealing with large and complex circuits especially arithmetic circuits. Furthermore, most of these approaches cannot provide concrete suggestions to remove Trojans. It is important to introduce efficient, scalable and fully automated test generation, Trojan localization and debugging framework.

1.1.2 Post-synthesized Non-functional Changes versus Security

After a design is implemented and validated in pre-silicon, the synthesized gate-level netlist will go through several non-functional changes for different purposes such as insertion of post-silicon design-for-test (DFT) and design-for-debug (DFD) facilities, clock tree insertion as well as dynamic power management [16] to ensure the performance and physical characteristics of the final fabricated chip. Most of these changes are done outside of the design house using untrusted vendors due to time-to-market as well as cost considerations. However, using untrustworthy vendors raise security concerns about the integrity of the design since hardware Trojans can be inserted during changes of the design. In order to trust an IP block, we have to make sure that the IP is performing exactly the expected functionality. Suppose that we have two versions of a design, one is a verified IP (specification) that is designed in-house and the other is an untrusted third-party IP (implementation) after performing non-functional transformations. Our goal is to detect whether an adversary has inserted hard-to-detect hardware Trojan during non-functional changes and has made undesired functional changes. For example, a design house may send their RTL design for synthesis or adding low-power features to a third-party vendor. Once the third-party IP comes back (after synthesis or other functionality-preserving transformations), it is crucial to ensure the trustworthiness of these IPs.

1.1.3 Control Flow Integrity Measurement

The security of an SoC can be compromised by exploiting the vulnerabilities of the nite state machines (FSMs) in the SoC controller modules through fault injection attacks. Fault injection poses a particularly serious threat. During fault attacks, an attacker injects faults to produce erroneous results and then analyzes these results to extract secret information from an SoC [10]. Over the past decade, fault injection attacks have grown from a crypto-engineering curiosity to a systemic adversarial technique [141]. However, most of the research on fault attacks are concentrated on analyzing the fault effects and developing countermeasures for fault injection on datapaths. Finite state machines in the control path are also susceptible to fault injection attacks, and the security of the overall SoC can be compromised if the FSMs controlling the SoC are successfully attacked. For example, it has been shown that the secret key of RSA encryption algorithm can be detected when FSM implementation of the Montgomery ladder algorithm is attacked using fault injection [129]. Therefore, it is also extremely important to identify and remove FSM vulnerabilities that facilitate fault attacks. These vulnerabilities may be introduced intentionally or unintentionally by designer mistakes, Trojan insertion, and traditional FSM design practices or by CAD tools during synthesis. It has been shown that synthesis tools can introduce security risks in the implemented FSM by inserting additional don't-care states and transitions [44, 102]. Authors in [44] proposed architectural changes in the FSM to address the vulnerabilities introduced by don't-care states and transitions.

1.1.4 Test Generation to Activate Trojans

Untrusted-third party IPs can come with malicious implants. Untrusted EDA tools, in-house rogue employees or the SoC integrator can also insert hard-to-detect Trojans in the original RTL design. We assume that to escape detection during different steps of verification/validation procedure, Trojans are designed in such a way that only a very rare set of input sequences can trigger them. In other words, Trojans are dormant during the normal execution, and activated under unusual (rare) conditions. Therefore, a smart adversary is likely to insert Trojans in RTL designs under rare branches which may reside in the unspecified functionality of the design. Otherwise, traditional simulation techniques using random or constrained-random tests can detect them, and the attacker's attempt would fail. Therefore, we need to have efficient test generation approaches to cover rare branches/assignments in pre-silicon designs in order to activate hidden hardware Trojans.

Post-silicon validation of an integrated circuit (IC) entails running tests on a fabricated, pre-production silicon to ensure that the design functions as expected under actual operating conditions and identify Trojans that have been missed during pre-silicon validation (e.g., RTL or software validation). Post-silicon validation is a highly complex activity, requiring elaborate planning, architectural support, and test development [106]. It is also an expensive activity accounting for more than 50% of the validation cost. Furthermore, post-silicon validation must succeed before mass production can begin.

Thus, effectiveness of post-silicon validation affects product launch, company revenues, profitability, and market positioning [140].

A fundamental problem in post-silicon validation is limited observability and controllability — only a few hundred among the millions of internal signals of an IC can be directly observed or controlled during silicon execution. This makes it difficult to diagnose bugs from observed failures of post-silicon tests, or even identify whether a test has passed, e.g., if the result of a test affects a signal which is not observable. In other words, it is difficult to determine whether the test has executed as expected.

To address this problem, it is critical that post-silicon tests be *observability-aware*, *i.e.*, produce results whose values can be reconstructed from the available observability. Unfortunately, this is difficult to achieve for several reasons. First, in an industrial IC development environment, observability architecture and (post-silicon) directed tests are developed independently and concurrently by different teams at different points of the design life-cycle. It is often impossible for test generation teams to account for silicon observability since the observability architecture may not have been fully developed at the time of test generation. Furthermore, it is difficult to employ automated tools for creating (additional) observability-friendly directed tests after the observability architecture has been defined. Creating the observability architecture entails analysis of the RTL models to identify traceable signals; these signals are then routed through appropriate hardware instrumentation to an observation point such as an output pin or memory [11, 83, 111]. On the other hand, analysis of RTL models directly to identify test generation is typically infeasible. RTL models tend to be large and complicated (typically millions of lines of code) making such analysis beyond the capacity of test analysis tools. RTL models may also contain functional errors or hardware Trojans. Moreover, there may be Trojans inserted during fabrication via an untrusted manufacturer. Indeed, a key reason for post-silicon directed testing is to identify such Trojans. Consequently, if one develops the directed tests through analysis of the RTL, then the fidelity of the tests as well as any inference made on their effects on observability, may become questionable. Therefore, it is essential to introduce efficient techniques for observability-aware post-silicon directed generation without utilizing RTL models.

1.1.5 Functional Coverage Analysis of Security Properties

Due to exponential growth of System-on-Chip (SoC) complexity, time-to-market reduction and huge gap between simulation speed and hardware emulation, there is a high chance that many Trojans escape from pre-silicon analysis and it affects the functionality of the manufactured circuit. To ensure the correct operation of the design, post-silicon validation is necessary. However, post-silicon validation is a bottleneck due to limited observability, controllability and technologies to cope with future systems [53, 101]. There is a critical need to develop efficient post-silicon validation techniques.

Currently there is no effective way to collect coverage of certain properties and events such as security properties directly and independently on silicon. Engineers need to assume that they will cover at least the same set of post-silicon coverage events as they cover with pre-silicon exercisers using accelerators/emulators [5]. However, we cannot be sure about the accuracy of these coverage metrics since silicon behaves differently than the simulated/emulated design mainly because of asynchronous interfaces. Moreover, because of time constraints, validation engineers are not able to hit all of the desired coverage events during pre-silicon validation or some coverage events are not activated enough. Therefore, they are seeking for an accurate and efficient way to know the coverage of desired events on silicon.

Assertions and associated checkers are widely used for design coverage analysis in pre-silicon validation to reduce debugging time. They can also be used in the form of coverage monitors to address controllability and observability issues as well as monitoring security events in post-silicon. However, every coverage monitor introduces additional area, power and energy overhead that may violate the design constraints. To address these limitations, there is a need for a framework to reduce the number of coverage monitors in post-silicon while utilize the existing post-silicon infrastructure such as debug facilities to enable functional coverage analysis. In the following section, we describe how these formal methods can be utilized to address the challenges introduced in this section.

1.2 Research Contributions

My research proposes novel techniques to address security validation challenges mentioned in Section 1.1. The goal of my research is to introduce efficient algorithms and tools based on formal methods to improve validation efforts at different stages of the design life cycle. Figure 1-4 also illustrates the comprehensive nature of my research that made the following fundamental contributions.

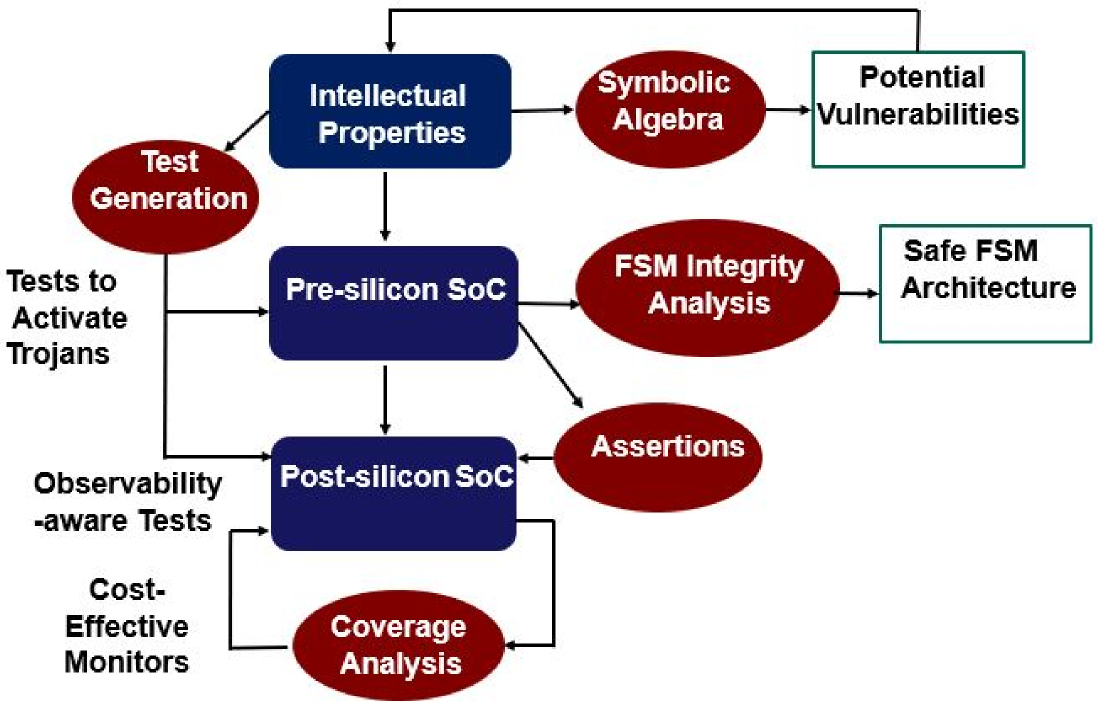


Figure 1-4. Comprehensive nature of my research.

Anomaly Detection and Correction in Arithmetic Circuits:

Existing arithmetic circuits verification approaches have focused on checking the equivalence between the specification of a circuit and its implementation. They use an algebraic model of the implementation using a set of polynomials F. The specification of an arithmetic circuit can be modeled as a polynomial f_{spec} using a decimal representation of primary inputs and primary outputs. The verification problem is formulated as mathematical manipulation of f_{spec} over polynomials in F. If the gate-level netlist has correctly implemented the specification, the result of equivalence checking is a zero polynomial; otherwise, it produces a non-zero polynomial containing primary inputs as variables (*remainder*). In this dissertation, I present a framework for directed test generation and automated debugging of datapath intensive applications using the remainder to detect and correct anomalies in the implementation.

Our method generates directed test vectors that are guaranteed to activate the malfunction. I consider gate misplacement or signal inversion that change the functionality of the design as our threat model. Next, I apply the generated tests, one by one, to find the faulty outputs that are affected by the existing anomaly. Regions that contribute in producing faulty outputs as well as their intersections are utilized for faster error localization. I show that certain errors manifest specific patterns in the remainder. This observation enables an automated debugging to detect and correct the source of error. Existence of a non-zero remainder as a result of applying equivalence checking between specification and implementation of an arithmetic circuit is a sign of a faulty implementation. However, there is no information about the number of existing errors in the implementation. There can be a single bug or multiple independent/dependent bugs in the design. The main question is that how to know the number of remaining bugs in the design and which algorithm should be used to fix them. In order to determine whether there is more than one bug in the implementation, I try to partition the remainder R into sub-remainders R_i first. I detect multiple dependent and independent bugs based on sub-remainders R_i.

Incremental Anomaly detection Depending on the location of the bug, the remainder generation can be challenging. The existence of a bug in the deeper stages of the design may make it extremely difficult to generate the remainder due to an explosion in the number of remainder terms (*term explosion effect*). The reason is that the faulty gate may introduce new terms during the intermediate steps of the specification polynomial's reduction. These extra terms are multiplied to polynomials of other gates and grow continuously until the remainder contains only primary inputs, leading to an explosion in the number of remainder terms.

I also present an incremental equivalence checking approach based on symbolic algebra that is done during several iterations. In each iteration, specification and implementation polynomials are updated based on the primary inputs' constraints, and Gröbner basis reduction is used to generate a remainder in order to define the result of the verification. If the verification results in a non-zero remainder, the implementation is Trojan-inserted. I use the generated remainder as well as inputs' constraints to detect and correct the source of the error. I also show that the order of constraints is important to efficiently debug the faulty implementation. Using the incremental verification approach coupled with different input ordering constraints enable us to efficiently generate remainder for a faulty design.

Trojan Localization using Symbolic Algebra: I propose a design-time formal method to localize and activate Trojans between two versions of a design. Suppose that there is a golden model of a design (specification), and a modified version (implementation) of it (after performing some non-functional changes such as doing synthesis, adding clock trees, scan chain insertion etc.). We would like to make sure that there is no hardware Trojan inserted to the design during the non-functional changes. In other words, our goal is to make sure that two versions of a design are functionally equivalent (nothing more, nothing less) and an adversary cannot hide hard-to-detect malicious modifications during design transformations. It is important to note that traditional equivalence checking techniques can lead to state-space explosion in the presence of specification and implementation of large and complex designs. Our approach is scalable since it uses polynomial based manipulation instead of Binary Decision Diagrams (BDD [110]).

I propose a formal method based on symbolic algebra to detect potentially malicious modifications in the implementation. Our method is based on extraction of functional polynomial [93] from gate-level IPs.

FSM Integrity Analysis in Controller Design: To detect vulnerabilities introduced by Trojan insertion as well as CAD tools, I propose an efficient, formal analysis framework based on symbolic algebra to find FSM vulnerabilities. The proposed method tries to find inconsistencies between the specification and FSM implementation through manipulation of respective polynomials. Security properties (such as a safe transition to a protected state) are derived using specification polynomials and verified against implementation polynomials. In case of a failure, the vulnerability is reported and a counter-example is generated. While existing methods can verify legal transitions, my approach tries to solve the important and non-trivial problem of detecting illegal accesses to the design states (e.g., protected states). I demonstrate the merit of my proposed method by detecting the vulnerabilities in various current FSM designs, while state-of-the-approaches failed to identify the security flaws.

Trojan Activation by Interleaving Concrete Simulation and Symbolic Execution: I propose a scalable directed test generation method to activate potential hardware Trojans in RTL models. The proposed approach is the

rst attempt in developing an automated and scalable technique to generate directed tests to activate hardware Trojans in RTL models. A threat model involving rare branches and rare assignments is considered in RTL designs. This threat model leads to the automated generation of security assertions. Concolic testing is utilized to generate tests to activate these security assertions. Interleaving concrete simulation with symbolic execution avoids state space explosion by exploring one execution path at a time in contrast to dealing with all possible execution paths at the same time (like conventional formal methods). Our experimental results demonstrate the effectiveness of our approach in activating hard-to-detect Trojans in large and complex Trust-Hub benchmarks.

Observability-aware Post-Silicon Test Generation: In this dissertation, I present a technique for observability-aware post-silicon directed test generation through analysis of pre-silicon design collaterals. Our key approach to overcome the scalability and relevant challenges mentioned above is to exploit more abstract transaction-level model (TLM) of a design to perform our analysis. TLM definitions are much more abstract, structured, and compact, compared to RTL, which permits effective application of exploration to identify high-quality directed tests. A key challenge is to map design functionality and observability between TLM and RTL so that the tests generated at TLM can be translated to effective, observability-aware tests for RTL. I discuss how to develop this mapping in practice. I provide case studies from a number of different design classes to demonstrate the flexibility and generality of our approach.

Cost-effective Synthesis of Security Assertions: Today's SoCs come with several built-in debug mechanisms such as trace buffers and performance monitors in order to enhance the design observability during post-silicon validation and reduce debugging efforts. Trace buffers record the values of a limited number of selected signals (typically less than 1% of all signals in the design) during silicon execution for specified number of clock cycles. The trace buffer values can be analyzed off-chip to restore the values of untraced signals. I present an approach to utilize the information that can be extracted from on-chip trace buffer in order to determine easy-to-detect functional coverage events using formal analysis. My trace-based coverage analysis enables the trade-off between observability and hardware overhead. The experimental results show that my approach can provide an order-of-magnitude reduction in design overhead without sacrificing functional coverage of security properties compared to when all assertions are synthesized.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 discusses the existing security validation approaches. Chapter 3 describes the algebraic preliminaries of formal verification methods based on symbolic algebra and Gröbner basis theory. Gröbner basis theory can be used in hardware Trojan detection as well as integrity analysis of controller designsChapter 4 describes anomaly detection procedure in combinational and arithmetic circuits. In Chapter 5, an incremental debugging approach is presented to improve the required effort for anomaly detection in arithmetic circuits. Chapter 6 introduces a technique for Trojan localization after non-functional changes of the gate-level netlist of the overall design. Chapter 7 describes a technique to identify and mitigate threats in controller designs. Chapter 8 presents a scalable directed test generation method to activate potential hardware Trojans in RTL models. Chapter 9 discusses the observability-aware test generation technique to improve Trojan activation in post-silicon. Chapter 10 presents an approach for getting coverage of post-silicon security properties using post-silicon debug infrastructures. Chapter 11 concludes this dissertation.

CHAPTER 2
RELATED WORK ON SECURITY VALIDATION

In this chapter, we review the existing security validation methods for SoCs in different stages. There has been plenty of research on trust validation in the IP level, pre-silicon, and post-silicon design. These methods focus on simulation based approaches, side-channel analysis, structural methods as well as formal approaches as shown in Figure 2-1. The remainder of this chapter surveys these approaches in detail.

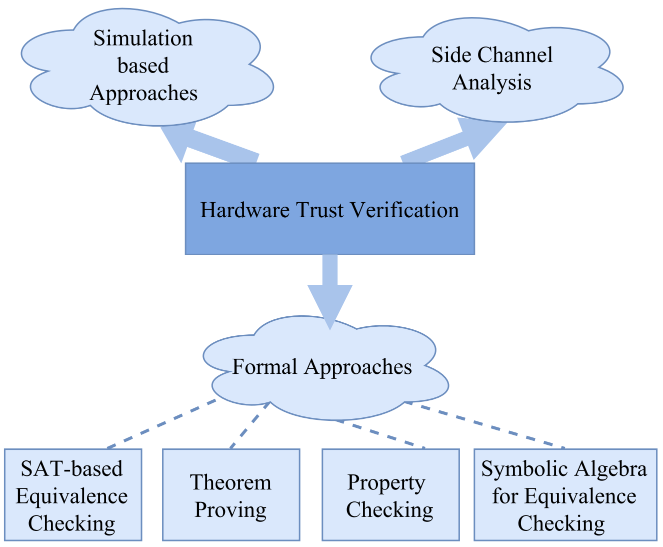

Simulation based Approaches
Side Channel Analysis
Hardware Trust Verification
Formal Approaches
SAT-based Equivalence Checking
Theorem Proving
Property Checking
Symbolic Algebra for Equivalence Checking

Figure 2-1. Hardware trust verification can be categorized in three major directions: (1) simulation based approaches, (2) side-channel analysis, and (3) formal approaches.

2.1 Simulation-based Trust Validation Approaches

Simulation-based approaches aim on generating tests to activate malicious modifications (hardware Trojans) and propagate the payload of the Trojan to primary outputs to check with the golden circuit. The difficulty of logic testing is to generate efficient tests to activate and propagate the effect of Trojans, which are stealthy enough to hide through the traditional manufacturing testing.

A major problem with design validation is that we do not know whether a Trojan exists, and how to quickly detect and fix it. We can always keep on generating random tests, in the hope of activating the Trojan; however, random test generation is neither scalable nor efficient when designs are large and complex.

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.

2.1.1 Functional Test Generation

In a recent case study [144], 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 [9, 144]. However, Trojans may exist in design that have 100% code coverage. The MERO (multiple excitation of rare occurrences) approach [30], proposed by Chakraborty et al., can generate high-quality tests to achieve very high activation and coverage rate for Trojans. A later work by Saha et al. [118] extended this approach by using genetic algorithm to further improve the quality of tests. Generating such directed tests is extremely difficult given the stealthiness of activation condition. Besides, this technique is only applicable to gate-level designs and does not guarantee whether the generated tests can activate the Trojans. Cruz et al. have proposed a test generation technique that combines the strength of model checking and ATPG for fast test generation [41]. Their approach partitions the design based on the scan chain. Constraints are generated for non-scan elements using model checking. These constraints as well as the scan elements are then given to ATPG for test generation. This approach is suitable only for partial scan-chain inserted designs. However, none of the existing techniques are scalable to activate and detect hidden Trojans. Moreover, 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 kind 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.

2.1.2 Statistical Methods

Statistical Trojan detection methods try to differentiate the Trojan-inserted circuit from the Trojan-free version using properties of known Trojans. FANCI is one such approach [135]. FANCI marks gates that weakly influence output signals as suspicious. Their proposed algorithm uses approximate truth table for each signal to infer its effect on the outputs. However, FANCI has a high false positive rate. A similar method named VeriTrust marks redundant logic gates as suspicious [142]. Initially, all gates that are not covered during verification phase are considered as suspicious nodes, and further analysis is carried out to confirm redundancy. FANCI and VeriTrust can detect only Trojans with always on or combinational type triggers (a trigger that depends only on current inputs). They cannot detect sequential Trojans, which is exploited by DeTrust benchmarks [143]. Hicks et al. proposed an approach for defeating Trojan based on unused circuit detection [65]. This method relies on the assumption that Trojan circuits will reside on unused portion of the circuit. However, their algorithm failed to detect Trojans that do not rely on unused circuits [125].

A score based classification method for detecting Trojan is discussed in [105]. The classification features are based on properties found from Trojans in Trust-Hub benchmarks [69]. Scores are given to nets for each of the matching features. Nets with score above a threshold are marked as Trojan nodes. Unfortunately, these features are too specific to Trust-Hub benchmarks and thus cannot be used as a generic detection method. A recent approach proposed by Salmani et al. [119] uses SCOAP[1] controllability and observability values to detect and isolate Trojan nodes. Controllability is defined as the number of primary inputs that must be manipulated to control a signal to a particular logic value. Observability is the number of primary input manipulations which is required to make a signal observable at the primary outputs. This method works using the assumption that Trojan nodes will have higher controllability/observability values to avoid detection. However, this approach will result in false positives in designs with partial scan chains. Benign signals that are not part of the scan chain will also have controllability/observability values similar to Trojans. Recently, a Trojan clustering approach based on signal correlation is proposed in [29]. However, this method is suitable for gate-level designs, and cannot be extended to RTL models for early detection.

2.1.3 Side-Channel Analysis

Existing techniques based on side channel analysis rely on the change of physical characteristics caused by the Trojan circuit - mostly in the form of current, power or delay [75, 90, 103]. If the side channel signature of a chip is different from the golden chip over a certain threshold, a Trojan is detected. For example, when a Trojan is partially or fully activated, it will have increased switching activity compared to Trojan free circuit. Wang et al. used this property to isolate Trojan [137]. MERS utilized test generation to improve the Trojan detection sensitivity [74]. Their approach selected the nodes with low transition probability as suspicious nodes. Then test vectors are applied in such a

[1] SCOAP: Sandia Controllability/Observability Analysis Program [60]

way that switching activity of these suspicious nodes become much higher than other nodes, increasing side-channel emission. The problem with side channel analysis is the presence of process variation and measurement noise. Side-channel based approaches face difficulty if the Trojan circuit is small. This is because difference in side channel signature due to the Trojan can be negligible compared to process variations. These methods also require Trojan free golden reference models. As side-channel analysis is carried out after fabrication, the chip may require re-spins if Trojan is detected. Thus, methods that can detect Trojan in an early design stage is highly desirable.

2.2 Security validation using Formal methods

Formal methods are promising in hardware validation as they evaluate the functionality and structure of the mathematical model to verify that the design correctly implements the functions described in the specification. Formal verification methods can be broadly classified in four groups: i) Satisfiability (SAT) solvers, ii) property checking using model checkers [19, 99], iii) theorem proving approaches [42], and iv) equivalence checking using decision graphs [25, 35, 110] and symbolic algebra [49]. In this section, we briefly discuss each of these methods and their applications for security validation. We focus on different formal methods to verify the design and detect un-desired functions (possibly Trojans).

2.2.1 Trust Validation using SAT Solvers

Given a Boolean formula, the satisfiability problem relies on finding Boolean values to the formula's variables such that the formula is evaluated to true. If such an assignment does not exist, the formula is called unsatisfiable. It means that any possible assignments to formula's variables force the formula to be false. The Boolean formula is constructed from AND, OR and NOT operators between various variables which can be either assigned to true or false. Many of the validation and debugging problems can be mapped to satisfiability problems. One of the applications is to check the equivalence between the specification of the circuit and its implementation using SAT-solvers. Figure 2-2 shows the equivalence checking using SAT-solvers. If the specification and implementation have the same functionality, the output of the XOR gate should always be false. If the output of XOR gate becomes true for any input pattern, it implies that the implementation and the specification do not have the same functionality for the same input pattern. In other words, if the circuit shown in Figure 2-2 is converted to conjunction normal form (CNF), a SAT-solver can be used to check the equivalence between the specification and implementation. If the SAT-solver reports unsatisfiable, we can conclude that specification and implementation are equivalent. Otherwise, they are not equivalence and the root of mismatch should be found. SAT solvers are excessively used for design validation [13, 14, 85].

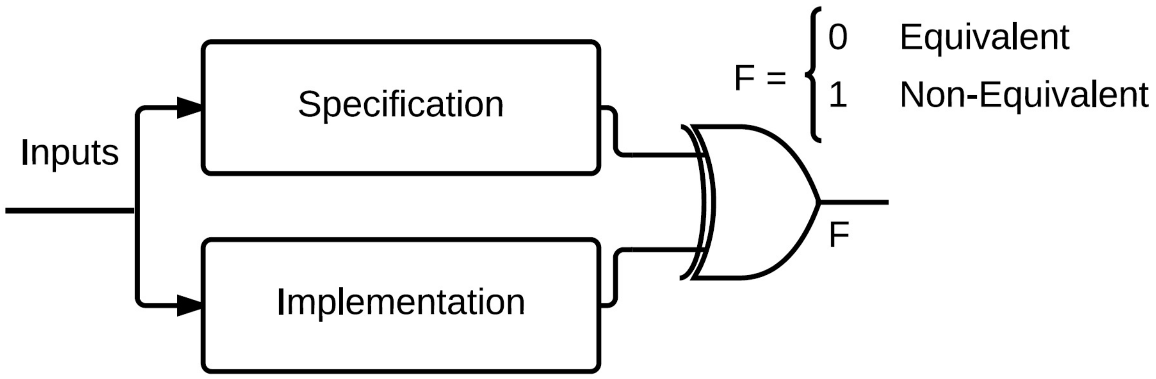


Figure 2-2. Equivalence checking using SAT solvers

Equivalence checking can be done using SAT-solvers to identify hardware Trojans [62]. If hardware Trojans exist in the implementation, the SAT-solver finds assignments to the internal variables to reveal the hidden Trojan. However, this method requires a golden model and suffers from scalability issues. The SAT-solver may encounter state explosion when the design is large, and the specification and the implementation significantly differ from each other.

Several works explore the existence of Trojans in unspecified functionality [55, 57]. Therefore, the Trojan does not alter the specification of the design, and existing statistical or simulation-based methods cannot identify the Trojan-inserted design [56]. Fern et al. propose a SAT-based technique to detect Trojans which exploit the design signals in their unspecified functionality to cause malfunction. Figure 2-3 shows a hardware Trojan in an unspecified functionality of a FIFO. The designer did not specify the functionality of the FIFO when the "*read_enable*" of the FIFO is not asserted. The attacker takes the advantage of the incomplete specification and inserts a malicious circuit to leak the secret information when the *read_enable* signal is not asserted. This kind of Trojans can be inserted in RTL code or at any high-level description of the design.

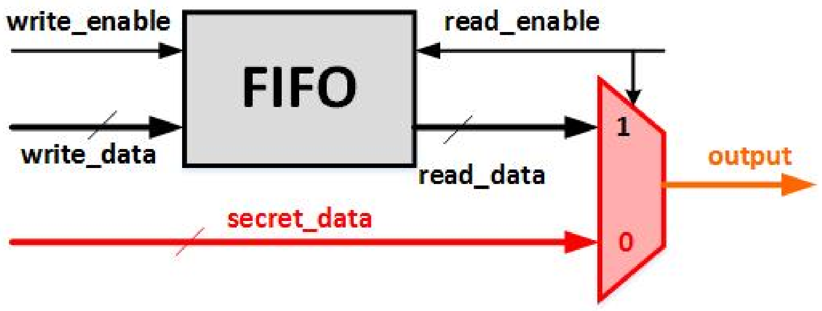


Figure 2-3. Trojan in unspecified functionality of a FIFO [56].

Fern et al. try to address unspecified Trojan detection where the Trojan targets information leakage [56]. Suppose that the function "$func$" is unspecified when internal signal "s" is under condition "C". Suppose that signal s can have two possible values: v_0 and v_1. Under condition C, Equation 2–1 should be unsatisfiable if the design is Trojan-free. Therefore, any assignment which make the Equation 2–1 satisfiable, it is a trace (counterexample) to detect the covert Trojan. For example, in the FIFO shown in Figure 2-3, $output$ signal should remain the same when the $read\_enable = 0$ ($C = read\_enable = 0$ and $s = output$).

$$C \wedge (func(s = v_0) \oplus func(s = v_1)) \tag{2–1}$$

To detect Trojans in an unspecified functionality of the design, pairs C and s should be identified. For any function in the design, several s and C pair can be found, and the process of marking the potential pairs is not automatic yet. For every pair (s, C), one

CNF formula is constructed and an SAT-solver (for Boolean values) or a Satisfiability Module Theory solvers (SMT-solvers) can be used to find the potential threats. The Trojan can be detected when the CNF formula is satisfiable. The success of this approach is dependent on the SAT-solvers and identifying (s, C) pairs. Moreover, the approach requires manual intervention.

SAT solvers [7, 97] have been also used to automatically localize and detect hardware Trojans (hard to detect bugs) in arithmetic circuits. Solving SAT problem results in finding suspicious functionality. These approaches are based on either inserting logic corrector components in the implementation [124], using abstraction and refinements [82, 117] or using Quantified Boolean Formula [97]. They model the circuit using CNF model, and a SAT solver is used to localize the sources of error. The success of these approaches is dependent on the performance of SAT solvers, and they fail for large and complex arithmetic circuits. There are some approaches based on Satisfiability modulo theory (SMT) solvers to find a counterexample and localize the source of problem in the faulty implementation [98, 126]. SMT solvers have been utilized to debug RTL designs [116]. Word-level MUXes are added to suspicious candidate signals and the resultant formula is solved by a word-level SAT solver. However, these methods are dependent on existence of traces of malicious activities. Moreover, these approaches suffer from the required manual interventions and cannot handle large designs. Furthermore, most of these approaches cannot provide concrete suggestions to fix Trojans.

2.2.2 Security Validation using Property Checking

Model checking is a famous technique in design verification which checks a design for a set of given properties. To solve the model checking problem, the design and the given properties are converted to a mathematical model/language, and all of the design's states are checked to see whether the given properties are satisfied. A class of model checkers is designed based on temporal logic formula [37]. The properties are described using Linear Temporal Logic (LTL) formulas to describe expected behaviors of the design.

The properties are checked using the model checkers. A model checker either proves the correctness of a given property over all of the possible behaviors of the design or find a counter-example when the property fails.

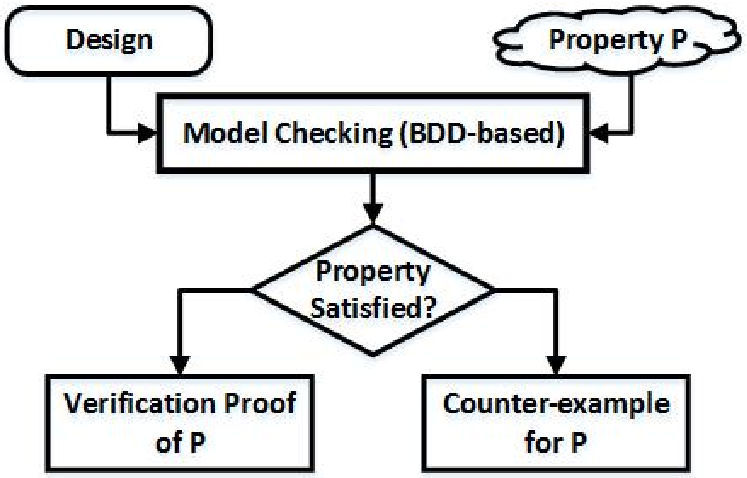


Figure 2-4. Verification using model checking.

A model checker tries all of the possible states of a design to prove a given property using a Binary Decision Diagram (BDD). However, the number of design states can be huge since every bit introduces two states in the design. For example, a 32-bit register can add 2^{32} states to the design state space. Although some techniques such as slicing, abstracting, etc. have been proposed [36, 133], state space explosion still is the largest limitation of using model checking in property verification. Bounded model checking (BMC) is introduced to overcome the amount of memory that a model checker requires for constructing and storing different states of a design [20]. BMC tries to find a counter-example in the first K cycles during execution. If a counter-example is found within K cycles, the property does not hold. Otherwise, K can be increased in the hope of finding a counter-example in upper bounds. BMC is not able to prove a property since it unrolls the circuit for a specified number of clock cycles. However, it can provide a statistical metric for a given property when the model checker fails (e.g. no counter-example can be found in K clock cycles). The BMC problem can be mapped to satisfiability problem, and SAT-solvers can be utilized to solve the problem. Therefore, the BMC addresses some of the state space explosion problems associated with BDDs in model checking. Figure 2-4 and Figure 2-5 shows the show the model checking and bounded model checking approaches, respectively. Clearly, bounded model checking cannot provide proof for property P. However, it can reveal when property P is violated within K clock cycles.

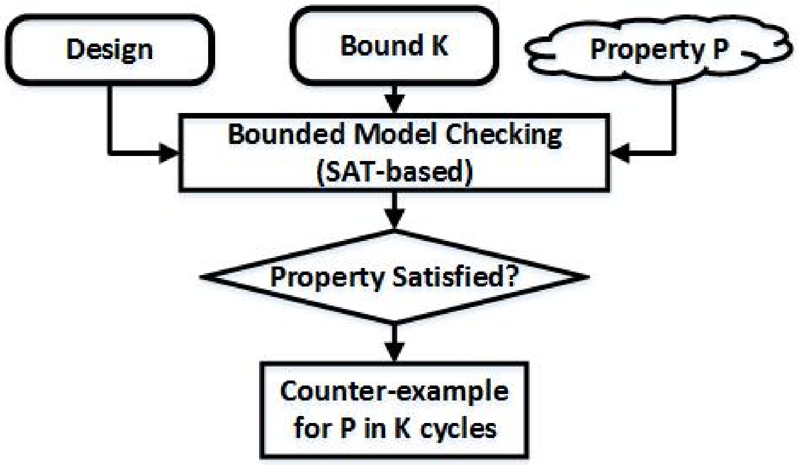

Design
Bound K
Property P
Bounded Model Checking (SAT-based)
Property Satisfied?
Counter-example for P in K cycles

Figure 2-5. Verification using bounded model checking.

Security properties describe the expected behaviors which a trustworthy design is required to follow. Model checkers can be used to ensure safety properties. An SoC designer and a third-party vendor can agree on certain security properties that the design should satisfy. When the design is sent to the SoC integrator, the SoC integrator converts the design to a formal description to check the security properties using a model checker. If all of the security properties are verified, the expected security behaviors are met. Rajendran et al. have proposed a Trojan detection technique which is based on using bounded model checking [113]. They have considered the threat model as an attempt to corrupt the critical data such as secret keys of a cryptographic design, and random numbers which are required by most of the cryptography algorithms or stack pointer of a processor. The assumption is that these critical data should be stored in some specific registers and accesses to these registers should be protected. In other words, the registers which contain critical data should be accessed through valid ways, and any undefined access to these registers is considered as a threat. The safe access conditions to these registers are formulated as properties (assertions), and a bounded model checker is utilized to find a counter-example when the security properties are violated.

Example 1: Suppose that the program counter (PC) register is considered as a critical data. The only valid ways to change the PC register is either using a reset signal (V_1), by CALL instruction which increments the PC register V_2 or using RET instruction which decrements the value of the PC register V_3. Otherwise, the PC register should keep its value. The safety property of PC register can be formulated as:

$$Safe\_PC\_change : assert \quad always \quad PC_{access} \notin \mathbb{V} = \{V_1, V_2, V_3\} \rightarrow PC_t = PC_{t-1})$$

When this property is fed into a bounded model checker alongside with the processor design, a counter-example is expected to be found whenever PC register or a part of it is changed using an unauthorized access. ■

Using model checking to find unauthorized access to secret and critical data of design is beneficial since the method does not require any golden model of the design. However, the success of this method is dependent on the SAT-engine (it may fail for large and complex designs) and precise definition of security properties which needs prior knowledge of all safe ways to access a critical register. The performance of the presented method can be further improved using an ATPG tool to ensure the trustworthiness of the assets for a large number of clock cycles. The property is synthesized as a circuit monitor, and it is appended to the original design. The ATPG tool is used to generate a test for a stuck-at-1 fault at the output of the monitor circuit. The counter-example can be found if the test can be generated. The success of this approach is dependent on the ATPG tools and complete definition of circuit monitors.

Researchers have proposed techniques based on formal methods to prove security-related properties that would be violated in the presence of Trojans. These methods are particularly effective for detecting Trojans inside cryptographic designs. One such method - GLIFT, looks for confidentiality and integrity property violation [73]. Confidentiality property requires that secret information never leaks to an unsecured domain and integrity property requires that untrusted data never enters the secured domain. Information flow is traced by assigning a taint bit to it. In another approach [112], a base property is used to detect information leakage which may imply the existence of a Trojan. The base property checks whether any input sequence exists such that it triggers secret information leakage to an observable point. The security properties check whether there is an input assignment (or a sequence of input assignments) I which triggers the leakage of secret data S to output ports or observable points (O) of the design.

$$\exists i \in I \rightarrow (S == O)$$

The property and formal description of the design are fed into a bounded model checker to find the possible leakage. However, the above-mentioned property has several challenges. If the secret information S contains n bits, the model checker needs to check 2^n different values. Checking all possible values may not be feasible when n is in the order of hundred (which is normal for encryption algorithms). The authors have proposed some refinements to limit the Trojan search to make information leakage detection feasible. However, the assumptions and refinement rules restrict the applicability of the solution to find different information leakage threats. Moreover, BMC may fail since the complexity of problem increases for each cycle of unrolling. Therefore, BMC works only for a certain number of clock cycles depending on the design size. When an adversary inserts a Trojan, which is triggered after a large number of clock cycles, this method cannot detect the Trojan.

Security property checking can be done in two general ways: (i) checking forbidden behaviors, and (ii) checking expected security properties. The malicious behavior of design is formalized and checked using model checkers in [114]. The method can be applied only for known Trojan types. Hasan et al. have proposed a hardware Trojan detection technique using LTL and computation tree logic (CTL) security properties to generate hardware Trojan monitors in order to improve the resiliency of hardware designs aginst malicious functionality [64]. The attacker is considered as an untrustworthy third party designer that can insert Trojans in the IP, and the defender is SoC integrator. The SoC integrator needs to formulate dangerous behaviors as security properties to perform vulnerability verification using model checkers. The generated counter-example, as well as the involved signals, are provided to the in-house designers to produce a guideline for efficient run-time security monitors.

Potential threats introduced from Electronic Design Automation (EDA) tools of the third party are considered in [109]. It is possible that an adversary modifies a design using non-transparent EDA tools such as synthesis tools. A synthesis tool may optimize some registers and unsafely modify the finite state machine (FSM). The authors have proposed a hardware Trojan detection technique which is based on property coverage analysis to ensure that a gate-level netlist is free from hardware Trojans inserted by synthesis tools. The proposed Trojan detection method is based on both security property checking as well as state coverage to mark suspicious unused circuit states. Figure 2-6 shows the different ways to insert Trojans in an FSM.

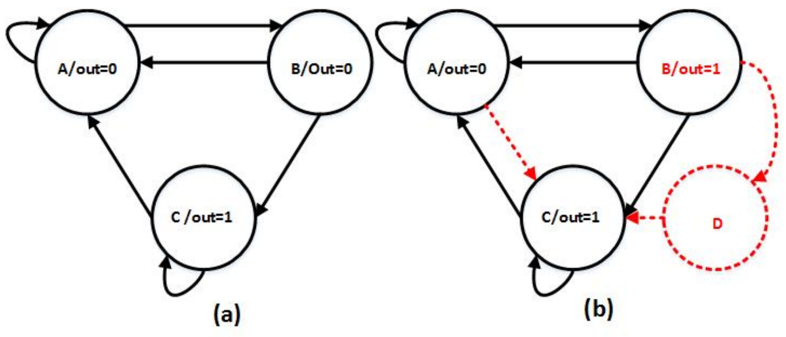


Figure 2-6. Trojans in a FSM: (a) A Trojan-free FSM, (b) Trojan can be inserted to a FSM using different ways: (i) changing the state output (e.g. state B), (ii) modification to state transitions (e.g. extra transition from state A to C), and (iii) adding extra states (e.g. state D) and transitions (such as state transitions $B \rightarrow D$ and $D \rightarrow C$) to FSM.

Example 2: Consider the FSM shown in Figure 2-6(a). Whenever the current state is A, the next state should be either A or B. The property can be formulated using a LTL formula as shown below:

$$assert \quad always \quad (cur\_state == A) \rightarrow X(next\_state = A || \quad next\_state = B)$$

Note that, X symbol shows the next cycle and $\rightarrow$ shows implication. ■

The success of using model checking-based approaches to detect hardware Trojans are highly dependent on the size of the design, SAT-engine and the quality of the provided properties. The model checker cannot guarantee inexistence of hardware Trojans. However, it can provide a trust level metric.

2.2.3 Theorem Provers for Trojan Detection

The attempt of proving a conjecture (logical statement) from a set of axioms and hypotheses is called theorem proving. Problems from different domains such as mathematics, hardware and software verification can be mapped to theorem proving. Automated theorem provers (ATP) are computer programs which try to prove given problems. They need appropriate and precise formulation of conjectures, axioms, and hypotheses in logical languages such as first-order logic or higher order logic. The language provides a formal description of the problem's statements; therefore, mathematical manipulation of statements is possible using ATPs. In other words, ATPs can show how the conjecture can be logically proved by following a set of related fact and statements. Figure 2-7 shows an overview of a theorem proving flow.

2.2.3.1 Secret Data Protection using Proof-Carrying Codes

Theorem provers are widely used in trust validation domain. Proof-carrying code (PCC)[104] has been proposed to provide a trust metric for a code often gathered from untrusted suppliers. The idea is that the code consumer should be able to confirm a set of pre-defined properties when the code producer delivers it. Using the PCC, the supplier is required to provide the formal proof of safety properties, and the consumer performs the

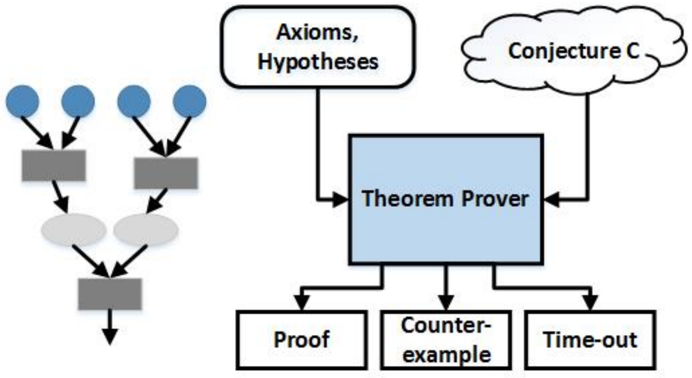


Figure 2-7. Validation using theorem Prover

proof validation to ensure the integrity of the code. Proofs are generated using theorem provers. Jin and Makris [77] have proposed an IP information tracking methodology which is based on proof carrying concept. The method is proposed to establish the trust between the untrusted IP vendor and the SoC Integrator. The IP vendor creates proof of security properties that was agreed upon with the SoC integrator. The design is instrumented with secrecy tags, and it is converted to a formal description. The design, secret tags as well as their formal descriptions are passed to the consumer as a package. The consumer formalizes the agreed security properties and regenerates the formal model of the HDL code to enable a formal property checker to validate proofs delivered by the producer. The assumption is that if an adversary inserts a hardware Trojan in the design to violate security properties, the proof validation will fail. Figure 2-8 shows the overall approach. The “Pass” result demonstrates the security preserving behavior of the delivered IP. However, the “Fail” result reveals the existence of malicious code to potentially leak secret information. Jin and Makris [77] have developed a set of rules to convert an HDL code to Coq formal language to automate formal model generation procedure.

Example 3: Suppose that the IP vendor is asked to deliver the implementation of DES algorithm [22] which is a data encryption algorithm. Our goal is to ensure that none of the secret data can be leaked through primary outputs. This requirement can be formalized as theorem “*Safe_DES*”. In the implementation of DES algorithm, the secret input "*KEY*", plaintext "*DesIn*" and internal key rounds "*KEY_Rounds*" are considered as secret information, and they should be protected. Therefore, they will be marked with security tags. Three axioms showing the secret character of *KEY*, *Des_In* and *KEY_Rounds* in all clock cycles are added to the formal logic which is used by the theorem prover. In the next step, three axioms, the *DES* implementation, as well as the *safe_DES* theorem, are converted to the Coq formal model. The theorem prover tries to prove the theorem by considering the formal behavior of the DES algorithm and axioms. If the proof can be generated, the *DES* algorithm is safe. Therefore, the design and the proofs can be delivered to the consumer. The user validates the proof using a property checker. ■

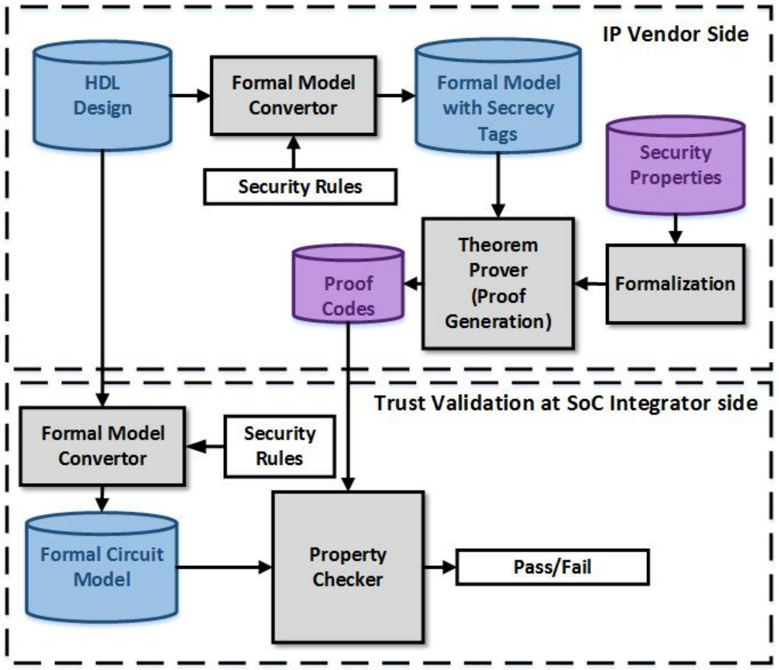


Figure 2-8. Information flow tracking based on proof-carrying codes.

A dynamic information tracking approach has been proposed in [79]. Similar to the statistical approach, the design, as well as security properties, are formalized and validated to detect unauthorized leakage of sensitive data. Unlike statistical scheme, all variables are assigned to a list of values showing their level of sensitivity during different clock cycles. Some updating rules are designed to change the sensitivity values of different variables over time. Two sensitivity lists are considered for data protection: (i) initial list, and (ii) the stable list, where values indicate stable sensitivity levels of circuit signals. The SoC integrator checks the content of both lists first to ensure the safe distribution of the secret data through the circuit. In the next step, proofs are validated.

A similar approach has been applied for trust validation of EDA tools [78]. The idea is that the trustworthy RTL design cannot guarantee the security of the IP cores. The gate-level netlist may be contaminated either by malicious implants or untrusted EDA tools. Jin [62] has proposed a framework to check the trust level of the produced gate-level netlists of synthesis tools. The framework consists of three major steps: (i) generates proof-carrying code based on the security properties of the trusted RTL code, (ii) validates proofs on the corresponding synthesized gate-level netlist, and (iii) measures the trust level of the EDA tools based on the result of the second step. The aforementioned approaches require flattening the design to check the integrity of the whole system. However, design flattening increases the complexity of formalization steps and proof generation phase, and limits the applicability of these approaches due to scalability problems. Similar to property checking approaches discussed in Section 2.2.2, the Trojan detection is dependent on the quality of security properties.

2.2.3.2 Integration of Theorem Provers and Model Checkers

The primary challenge in using proof-carrying code approach is to measure the trust of RTL, or gate-level code is the scalability. SoC designs are usually large and complex, and proofs cannot be easily constructed as the size of the design is increased. Proof validation is also very time consuming for large designs. To address these issues, an integrated approach has been proposed [63]. The main idea is to combine interactive theorem provers with model checkers to check security properties. The hierarchical structure of the SoC design is considered as a choice of design partitioning to reduce the verification efforts and address the scalability issues. The security properties are formalized as theorems and theorems are decomposed into several lemmas each related to one partition of the design. The lemmas are converted to assertions, and a model checker is used to validates the assertions. Each assertion is called a sub-specification. Sub-Specifications are selected in a way that they are dependent on each other. If all of the partitions satisfies security lemmas, we need to use the theorem prover to validate the security of whole design using checked lemmas. If the security theorem can be proven, the whole system is considered safe.

This method distributes the proof construction to overcome the scalability problem. However, the success of this approach is dependent on the model checking engine. In some cases, it may not be possible to validate the lemmas because of the model checkers limitations.

2.2.4 Trojan Detection using Symbolic Algebra

Equivalence checking is another way of formally proving a circuit is Trojan free. Such approaches require a golden specification to verify if it is equivalent to the implementation. Trojan inserted implementation will demonstrate functionality outside of the specification. However, traditional equivalence checking techniques suffer from state explosion issue. For example, equivalence checking has been done using SAT solvers and industrial tools such as Formality [71] traditionally. However, these methods are promising when the specification and implementation structures are similar such as between RTL (pre-synthesis) and gate-level (post-synthesis) models. However, existing methods can lead to state space explosion when complex SoCs are involved with significantly different (lack of structural or FSM-level similarity) specification and implementation.

A promising direction to address the state space explosion problem of equivalence checking of hardware design is to use methods based on symbolic computer algebra. Symbolic algebraic computation refers to the application of mathematical expressions and algorithmic manipulations methods to solve different problems. Symbolic algebra has received attention because of its applicability in equivalence checking of hardware designs. There are equivalence checking methods based on symbolic algebra that are successful to detect deviations from the specification for combinational circuits specially arithmetic circuits [46, 50, 62, 121]. These method maps the equivalence checking problem to ideal membership testing and solves the problem using Gröbner Basis theory. They express both specification and implementation as polynomials, and reduce the specification polynomials over a subset of implementation polynomials. If both are same, then the reduction procedure should result in a zero remainder. Any non-zero remainder indicates deviation from the specification. Such methods not only detect the Trojan existence, but also can isolate the Trojan circuit and generate test vector for activation.

Every extra, incorrect or missing components can threaten the security of the design. Ghandali et. al [58] proposed an automated debugging approach based on symbolic computer algebra which scans the entire implementation to find and fix the bug in the design. This approach suffers from scalability concerns. It has been shown that the remainder can be beneficial in root causing the threat [52]. The threat model is considered any deviation from the expected functionality. The remainder can be utilized to generate tests to activate the Trojan. If there are more than one malicious functionality in the implementation, the remainder will be affected by all of them. Therefore, each assignment that makes the remainder non-zero activates at least one of the existing faulty scenarios. The generated tests and the remainder's patterns can be used to localize malicious scenarios.

2.3 Summary

This chapter presented existing techniques for hardware security validation. It outlined prior efforts in test generation, side-channel based techniques as well as formal approaches for detecting hardware Trojans.

CHAPTER 3
BACKGROUND: VERIFICATION USING SYMBOLIC ALGEBRA

A promising direction to address the state space explosion problem in trust validation 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). In this chapter, Gröbner basis theory [39] is briefly described. Next, the application of Gröbner basis theory for security verification of integer arithmetic circuits is presented. We will describe how *Gröbner basis* theory can be used in hardware Trojan detection as well as integrity analysis of controller designs in Chapters 4, 5, 6, and 7.

3.1 Gröbner Basis Theory

Let $M = {x_1}^{\alpha_1}{x_2}^{\alpha_2}...{x_n}^{\alpha_n}$ be a monomial and $f = C_1M_1 + C_2M_2 + ... + C_tM_t$ be a polynomial with $\{c_1, c_2, ..., c_t\}$ as coefficients and $M_1 > M_2 > ... > M_t$. Monomial $lm(f) = M_1$ is called leading monomial and $lt(f) = C_1M_1$ is called leading term of polynomial f. Let $\mathbb{K}$ be a computable field and $\mathbb{K}[x_1, x_2, ..., x_n]$ be a polynomial ring in n variables. Then $< f_1, f_2, ..., f_s >= \{\sum_{i=1}^{n} h_if_i : h_1, h_2, ..., h_s \in \mathbb{K}[x_1, x_2, ..., x_n]\}$ is an ideal I. The set $\{f_1, f_2, .., f_s\}$ is called generator or basis of ideal I. If $V(I)$ shows the affine variety (set of all solution of $f_1 = F_2 = ... = f_s = 0$) of ideal I, $I(V) = \{f_i \in \mathbb{K}[x_1, x_2, ..., x_n] : \forall v \in V(I), f_i(v) = 0\}$. 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 $G = \{g_1, g_2, ..., g_t\}$ is called Gröbner basis of ideal I, if $\forall f_i \in I, \exists g_j \in G : lm(g_j)|lm(f_i)$.

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 $lt(f_i) = C_1M_1$ (which is non-zero) is divisible by $lt(g_i)$ and r is the remainder ($r = f_i - \frac{lt(f_i)}{lt(g_j)}.g_j$). 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 $f_i \xrightarrow{G}_+ r$.

Algorithm 1: Buchberger's algorithm [27]

```
1: Input: ideal $I =< f_1, f_2, ..., f_s >\neq \{0\}$, initial basis $F = \{f_1, f_2, ..., f_s\}$
2: Output: Gröbner Basis $G = \{g_1, g_2, ..., g_t\}$ for ideal I
3: $G = F$
4: $V = G \times G$
5: while $V \neq 0$ do
6:    for each pair $(f, g) \in V$ do do
7:       $V = V - (f, g)$
8:       $Spoly(f, g) \rightarrow_G r$
9:       if $r \neq 0$ then
10:         $G = G \cup r$
11:         $V = V \cup (G \times r)$
12:      end if
13:   end for
14: end while
15: Return: set $G$
```

The set G is Gröbner basis ideal I, if $\forall f \in I, f_i \xrightarrow{G}_+ 0$. Gröbner basis can be computed using Buchburger's algorithm [27]. Buchburger algorithm is shown in Algorithm 1. It makes use of a polynomial reduction technique named S-polynomial as defined below.

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

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:

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

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

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

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

It is obvious that S-polynomial computation cancels leading terms of the polynomials. As shown in Figure 1, Buchbergers algorithm first calculates all S-polynomials (lines 4-6 of Fig. 2) and then adds non-zero S-polynomials to the basis G (line 8). 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 [27]. When the size of F increases, the verification process may be very slow or in the worst-case may be infeasible.

Buchberger algorithm is computationally intensive and it may affect the performance drastically. It has been shown in [26] that if every pair (f_i, f_j) that belongs to set $F = \{f_1, f_2, ..., f_s\}$ (generator of ideal I) has a relatively prime leading monomials $(lm(f_i).lm(f_j) = LCM(lm(f_i).lm(f_j)))$ 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 Figure 3-1. The major computation steps in Figure 3-1 are outlined below:

- Assuming a computational field $\mathbb{K}$ and a polynomial ring $\mathbb{K}[x_1, x_2, ..., x_n]$ (note that variables $\{x_1, x_2, ..., x_n\}$ are subset of signals in the gate level implementation), a polynomial $f_{spec} \in \mathbb{K}[x_1, x_2, ..., x_n]$ representing specification of the arithmetic circuit can be derived.
- Map the implementation of arithmetic circuit to a set of polynomials that belongs to $\mathbb{K}[x_1, x_2, ..., x_n]$. The set F generates an ideal I. Note that according to the field $\mathbb{K}$, some vanishing polynomials that constructs 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

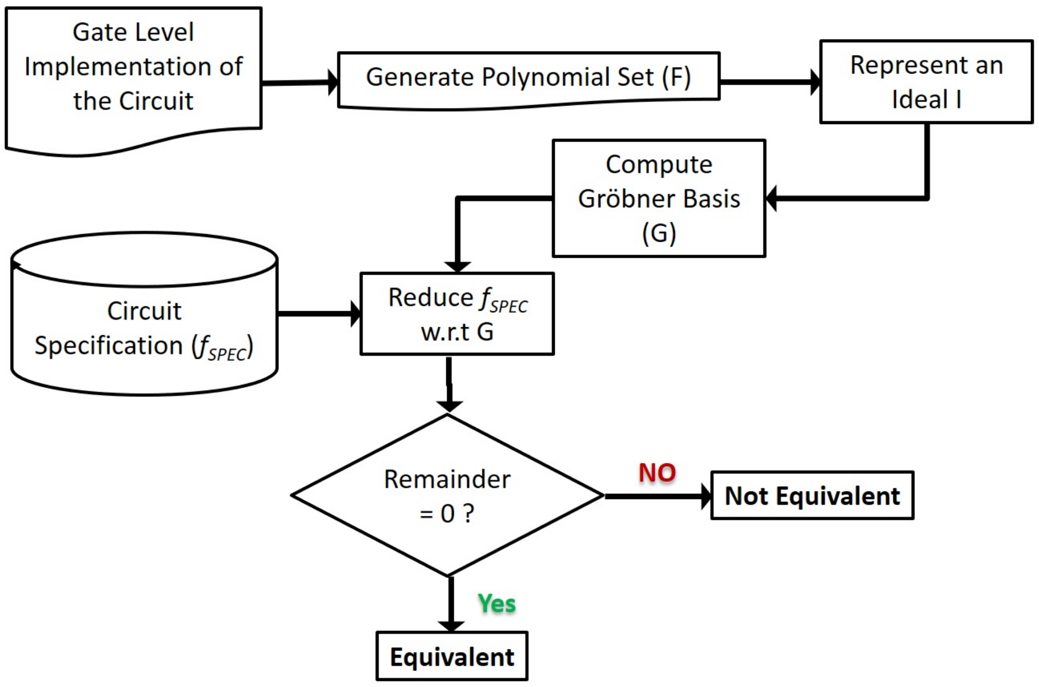

Gate Level Implementation of the Circuit
Generate Polynomial Set (F)
Represent an Ideal I
Compute Gröbner Basis (G)
Circuit Specification (f_{SPEC})
Reduce f_{SPEC} w.r.t G
Remainder = 0 ?
NO
Not Equivalent
Yes
Equivalent

Figure 3-1. Equivalence checking flow

combinational arithmetic circuits are acyclic, the topological order of the signals in the gate level implementation can be used.

- 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 $f_{spec} \xrightarrow{G}_{+} r$. 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 [94], Mastrovito multiplication and Montgomery reduction [84] which are critical part of cryptosystems. In order to apply the method of Figure 3-1 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 [91] has extracted circuit polynomials by converting each gate to a polynomial and SINGULAR [61] has been used to do the $f_{spec} \xrightarrow{G}_{+} r$ 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 has been proposed in [92]. The cost of $f_{spec} \xrightarrow{G}_{+} r$ 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 $\mathbb{Z}[x_1, x_2, , x_n]/2^N$ in [50]. 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 $\mathbb{Z}[x_1, x_2, , x_n]/2^N$ contain carry chain, the number of polynomials can be very large). Therefore, the number of circuit polynomials are decreased. In order to expedite the $f_{spec} \xrightarrow{G}_{+} r$ computation, the polynomials are represented by Horner Expansion Diagrams. The reduction computation is implemented by sequential division. The verification of arithmetic circuit over ring $\mathbb{Z}[x_1, x_2, , x_n]/2^N$ up to 128 bit can be efficiently performed using this method. An extension of this method has been presented in [47] 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 [92], 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.

3.2 Verification of Arithmetic Circuits

Most of the traditional verification and debugging tools of arithmetic circuits are based on techniques such as simulation, binary decision diagrams (like BDDs,*BMD [25]) and SAT solvers [7, 97]. However, all of these approaches suffer from state space explosion while dealing with large and complex circuits especially arithmetic circuits. Furthermore, most of these approaches cannot provide concrete suggestions to remove Trojans. It is important to introduce efficient, scalable and fully automated verification framework.

Computer symbolic algebra is employed for equivalence checking of arithmetic circuits to address the limitations of traditional approaches. 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 is high. These bugs may cause unwanted operations as well as security problems like leakage of secret key [21]. Thus, verification of arithmetic circuits is very important.

A set of polynomials $\mathbb{F} = \{F_1, F_2, ..., F_n\}$ which are defined over a field generates an ideal I where $I =< F_1, F_2, ..., F_n >$. Set $\mathbb{F}$ is called basis or generator of Ideal I. Generally, Ideal I can have several bases. One of the basis is called Gröbner Basis G which can be derived from Buchberger's algorithm [26]. The main characteristic of Gröbner Basis is the ability to solve the ideal membership problem [27]. In other words, if we want to check whether polynomial f resides in ideal I, f can be reduced over setting G (reduction can be done using polynomial sequential division regarding a specific order). If the result of reduction is equal to zero polynomial, f belongs to ideal I. Otherwise f does not reside in the ideal I [38].

The arithmetic circuit equivalence checking problem formulation starts with converting the design specification to a polynomial f_{spec} which represents the word-level abstraction of arithmetic circuits functionality using primary inputs and primary outputs as variables. For example, the specification of a n-bit adder with primary inputs $A = \{a_0, a_1, ..., a_{n-1}\}$ and $B = \{b_0, b_1, ..., b_{n-1}\}$ and primary output $Z = \{z_0, z_1, ...z_n\}$ can be formulated as $Z = A + B$ or can be written as $(2^n.z_n + ... + 2.z_1 + z_0) - ((2^{n-1}.a_{n-1} + ... + 2.a_1 + a_0) + (2^{n-1}.b_{n-1} + ... + 2.b_1 + b_0)) = 0$ where $\{a_i, b_i, z_i\} \subset \{0, 1\}$.

The functionality of logic gates (such as AND, OR, XOR, NOT and buffer) can be represented by polynomials such that the input and output signals of gates act as variables of the corresponding polynomial. Each variable x_i which appears in a circuit polynomial, belongs to $\mathbb{Z}_2$ where $({x_i}^2 = x_i)$. Equation 3–1 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 Equation3–1.

$$\begin{aligned}
&z_1 = NOT(a) \rightarrow z_1 = 1 - a,\\
&z_2 = AND(a, b) \rightarrow z_2 = a.b,\\
&z_3 = OR(a, b) \rightarrow z_3 = a + b - a.b,\\
&z_4 = XOR(a, b) \rightarrow z_4 = a + b - 2.a.b
\end{aligned} \tag{3–1}$$

A gate-level netlist of a circuit can be modeled as a set of polynomials $\mathbb{F}$ by modeling each gate as a polynomial. Suppose that we want to make sure an arithmetic circuit implements correctly its specification. In other words, we want to verify that there are no functional errors in the arithmetic circuit. The equivalence checking starts with consecutively reducing the f_{spec} over implementation polynomials ($\mathbb{F}_{imp}$) until either zero remainder or a remainder that contains only primary input variables are reached. If the remainder is zero, it shows that the arithmetic circuit performs the exact specification.

However, the non-zero remainder shows that the implementation is not trustworthy and there are some malfunctions.

Example 2: Suppose that we want to verify the functional correctness of a full-adder implementation shown in Figure 3-2. The specification can be formulated as: $(2.C_{out} + S - (A + B + C_{in}))$ and each gate in the implementation can be modeled as a polynomial based on Equation 3–1. The topological order of the circuit (since the circuit is acyclic) is chosen for reduction as $C_{out} > \{S, n_3\} > \{n_2, n_1\} > \{A, B, C_{in}\}$. The reduction starts from the most significant primary output and ends at primary inputs. Variables in the curvy brackets have the same order and they can be reduced in one iteration. Equation 3–2 shows the reduction process. It can be seen that the final result (remainder) is a non-zero polynomial and implementation is not trustworthy. The reader can verify that the remainder would be zero if the NAND gate is replaced with an AND gate.■

$$\begin{aligned}
&step_0 : 2.C_{out} + S - A - B - C_{in}\\
&step_1 : S - 2.n_3.n_2 + 2.n_3 + 2.n_2 - A - B - C_{in}\\
&step_2 : 2.n_2.n_1.C_{in} - 4.n_1.C_{in} + n_1 - A - B + 2\\
&step_3(remiander) : 8.A.B.C_{in} - 4.A.C_{in} - 4.B.C_{in} - 2.A.B + 2
\end{aligned} \tag{3–2}$$

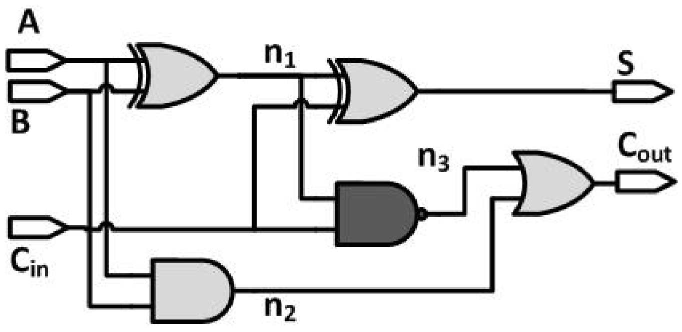


Figure 3-2. Faulty gate-level netlist of a full-adder. The NAND gate should be replaced by an AND gate to correct the bug.

3.3 Summary

In this chapter, we discussed Gröbner basis reduction theory that can be used for the basis of remainder generation, test generation for trust verification, as well as automated debugging described in subsequent chapters.

CHAPTER 4
ANOMALY DETECTION AND CORRECTION IN ARITHMETIC CIRCUITS

Optimized and custom arithmetic circuits are widely used in embedded systems such as multimedia applications, cryptography systems, signal processing and console games. Debugging of arithmetic circuits is a challenge due to increasing complexity coupled with non-standard implementations. Existing equivalence checking techniques produce a remainder to indicate the presence of a potential bug. However, bug localization remains a major bottleneck. Simulation-based validation using random or constrained-random tests are not effective and can be infeasible for complex arithmetic circuits. In this chapter, we present an automated test generation and bug localization technique for debugging arithmetic circuits. This chapter makes two important contributions. Figure 4-1 shows an overview of our proposed framework. We propose an automated approach for generating directed tests by suitable assignments of input variables to make the reminder non-zero. There can be several possible assignments that make remainder non-zero; each of these assignments is essentially a test vector that is guaranteed to activate the bug. The generated tests are guaranteed to activate the unknown bug. We apply the generated tests, one by one, to find the faulty outputs that are affected by the existing bug. Regions that contribute in producing faulty outputs as well as their intersections are utilized for faster bug localization. We also propose a bug detection and correction technique by utilizing the patterns of remainder terms as well as the intersection of regions activated by the generated tests. We show that certain bugs manifest specific patterns in the remainder. This observation enables an automated debugging to detect and correct the source of error. Our experimental results demonstrate that the proposed approach can be used for automated debugging of complex arithmetic circuits.

In this chapter, we also present algorithms to detect and correct multiple independent bugs. The proposed approach partitions the remainder R in order to generate several sub-remainder R_i such that each of them is responsible for a single bug b_i. In the next step, we generate directed tests from each sub-remainder to localize the source of error. We apply the single bug detection algorithm on each of sub-remainders in order to correct multiple unknown independent bugs. If the remainder cannot be partitioned in sub-remainders, and single bug detection approach cannot report any source of error for a faulty design, we can conclude that there are multiple dependent bugs in the implementation (the bugs that have effects on each other). In this chapter, we also propose an algorithm to detect and correct two dependent unknown bugs. The complexity of the algorithm grows linearly with the number of suspicious gates (suspicious gates can be obtained as bug localization phase).

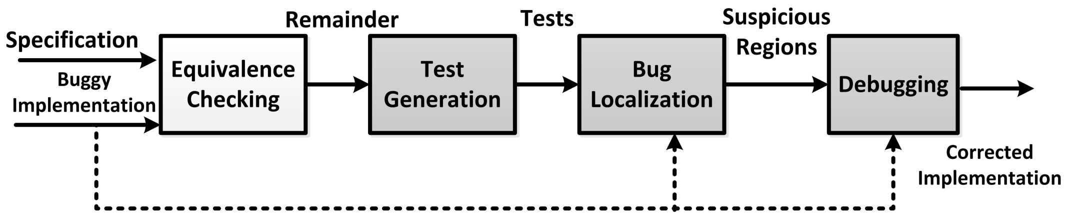


Figure 4-1. Overview of our automated debugging framework. It consists of three important steps: test generation, bug localization, and automated debugging of arithmetic circuits.

Figure 4-2 shows different scenarios for a buggy implementation. Figure 4-2 (a) illustrates the case when only one bug exists in the implementation. Figure 4-2(b) shows the presence of two bugs which do not share input cones (independent bugs). We describe how to fix one or more independent bugs in Section 4.1 and Section 4.2.1, respectively. We present algorithms to detect and correct multiple independent bugs. In many cases, bugs may share input cones as shown in Figure 4-2 (c). In this chapter, we also propose an algorithm to detect and correct multiple dependent unknown bugs in Section 4.2.2. Generally, a buggy implementation can contain any combination of independent and dependent bugs as shown in Figure 4-2 (d).

Figure 4-3 shows different steps of our proposed debugging approach to detect and correct multiple bugs for various scenarios depicted in Figure 4-2. In Section 4.1, we present a single bug detection and correction algorithm. In order to determine that

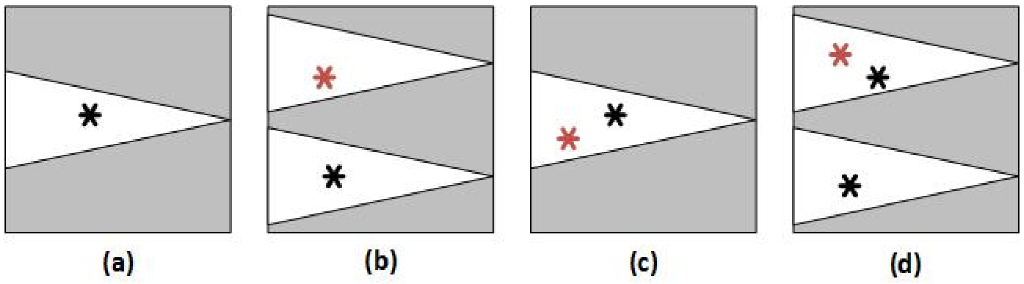


Figure 4-2. Relative bugs' locations and their corresponding input cones of influence.

whether there is more than one bug in the implementation, we try to partition the remainder R into sub-remainders R_i first. If remainder can be partitioned successfully into n sub-remainders, we can conclude that there are at least n independent bugs in the implementation as we discussed in Section 4.2.1. Algorithms in Section 4.1 are used over each sub-remainder R_i to detect and correct each bug. However, if a single bug cannot be found for remainder R_i, there are multiple dependent bugs which construct the sub-remainder R_i. Therefore, we try to find a single bug corresponding to remainder R_i first. If we can find such a bug, the bug will be fixed. Otherwise, we try the proposed the algorithm of Section 4.2.2 to find dependent bugs responsible for sub-remainder R_i. The procedure will be repeated for all of the sub-remainders. To the best of our knowledge, our proposed method is the first attempt to automatically detect and correct multiple dependent/independent bugs in arithmetic circuits.

The remainder of the chapter is organized as follows. Section 4.1 discusses our framework for directed test generation and bug localization/detection approach. Section 4.2 describes our debugging approach to detect and correct multiple bugs. Section 4.3 presents our experimental results. Finally, Section 4.4 concludes the chapter.

4.1 Automated Debugging using Remainders

Our framework uses the remainder that is generated by equivalence checking. If the remainder is a non-zero polynomial, it means that the implementation is buggy; however, the source of the bug is unknown.

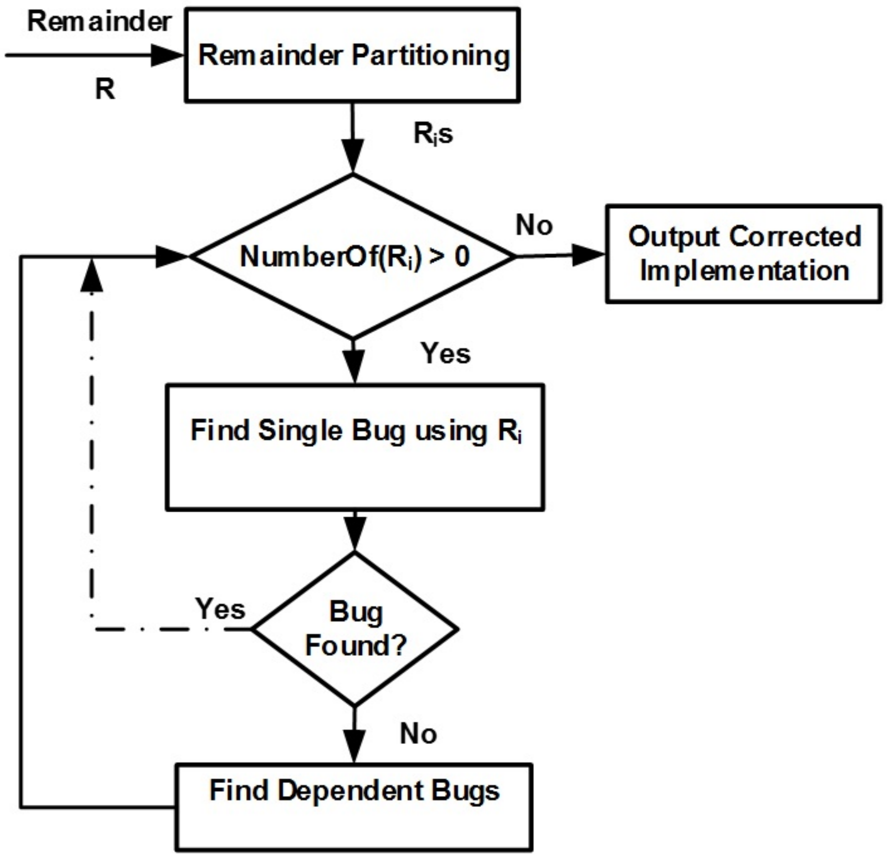


Figure 4-3. Overview of different steps of our proposed debugging framework. Independent bugs are detected and corrected using the first looop with dotted line as described in Section 4.2.1. Debugging of dependent bugs are discussed in Section 4.2.2.

Example 1: Consider a 2-bit multiplier with gate-level netlist shown in Fig. 4-4. Suppose that, we deliberately insert a bug in the circuit shown in Fig. 4-4 by putting the XOR gate with inputs (A_0, B_0) instead of an AND gate. The specification of a 2-bit multiplier is shown by f_{spec}. The verification process starts from f_{spec} and replaces its terms one by one using information derived from the implementation polynomials as shown in Equation 4–1. For instance, term $4.Z_2$ from f_{spec} 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 Equation 4–1. Clearly, the remainder is a non-zero polynomial and it reveals the fact that the implementation is buggy. ■

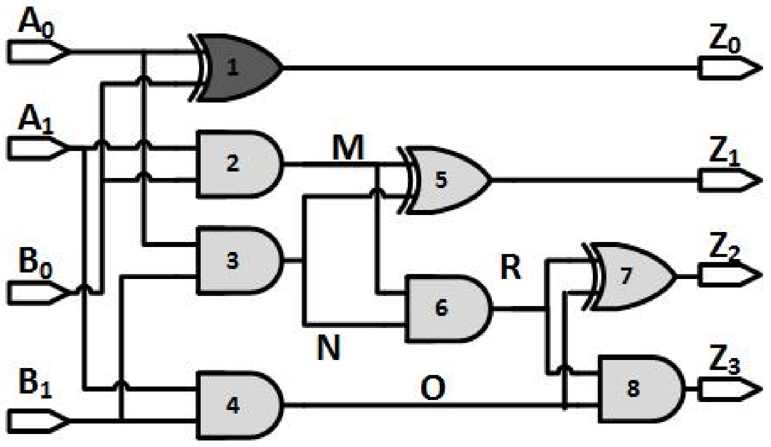


Figure 4-4. Faulty gate-level netlist of a 2-bit multiplier

$$
\begin{aligned}
&f_{spec} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&step_1 : 4.R + 4.O + 2.z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&step_2 : 4.O + 2.M + 2.N + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&step_3(remiander) : 1.A_0 + 1.B_0 - 3.A_0.B_0
\end{aligned}
\tag{4–1}
$$

Our approach takes the remainder and the buggy implementation as inputs and tries to find the source of error in the implementation and correct it. As shown in Fig. 4-1, our debugging framework has three important steps. First, we use the remainder to generate directed tests to activate faulty scenarios. Next, we try to localize source of the bug by leveraging the generated tests. Finally, we use an automated correction technique to detect and correct the existing bug which resides in the suspicious area. We describe each of these steps in detail in the following sections.

4.1.1 Directed Test Generation

It has been shown that if the remainder is zero, the implementation is bug-free [138]. Thus, when we have 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. Remainder is a polynomial with Boolean/integer coefficients. It contains a subset of primary inputs as its variables. Our approach takes the remainder and finds all of the assignments to its variables such that it makes the decimal value of the remainder non-zero. As shown in Example 1, the remainder may not contain all of the primary inputs. As a result, our approach may use a subset of primary inputs (that appear in the remainder) to generate directed tests with *don't cares*. Such assignments can be found using a SMT solver by defining Boolean variables and considering signed/unsigned integer values as total value of the remainder polynomial ($i \neq 0 \in \mathbb{Z}, check(R = i)$). The problem of using SMT solver is that for each i, it finds at most one assignment of the remainder variables to produce value of i, if possible. We implemented an optimized parallel algorithm to find all possible assignments which produce non-zero decimal values of the remainder. Algorithm 2 shows the details of our test generation algorithm. The algorithms gets remainder R polynomial and primary inputs (PI) in the remainder as inputs and feeds binary values to PIs (s_i) and computes the total value of a term (T_j). 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).

Algorithm 2: Directed Test Generation Algorithm

```
1: **Input:** Remainder, R
2: **Output:** Directed Tests, Tests
3: **for** different assignments $s_i$ of PIs in R **do**
4:    **for** each term $T_j \in R$ **do**
5:       **if** ($T_j(s_i)$) **then**
6:          $Sum+ = C_{T_j}$
7:       **end if**
8:    **end for**
9:    **if** ( Sum != 0 ) **then**
10:      $Tests = Tests \cup s_i$
11:   **end if**
12: **end for**
13: **Return:** Tests
```

Example 2: Consider the faulty circuit shown in Fig. 4-4 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 ($A_1 = 0, B_0 = 1$). 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 are shown in Table 7-1.

Table 4-1. Directed tests to activate fault shown in Fig. 4-4

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

4.1.2 Bug Localization

So far, we know that the implementation is buggy and we have all the necessary tests to activate the faulty scenarios. Our goal is to reduce the state space in order to localize the error by using tests generated in the previous section. The bug location can be traced by observing the fact that the outputs can possibly be affected by the existing bug. The proposed methodology is based on simulation of test cases.

We simulate the tests and compare the outputs with the golden outputs and keep track of faulty outputs in set $E = \{e_1, e_2, .., e_n\}$. Each e_i denotes one of the erroneous outputs. To localize the bug, we partition the gate-level netlist such that fanout free cones (set of gates that are directly connected together) of the implementation are found. Algorithm 3 shows the procedure of partitioning of gate-level netlist of a circuit.

Each gate that its 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 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 3 chooses one fanout gate from L_{fo} and goes backward

Algorithm 3: Fanout-free cone finder algorithm

```
1: **Input:** Circuit C, Fanout list, $L_{fo}$
2: **Output:** Fanout-free regions, Cones
3: **for** Each fanout gate $g_i \in L_{fo}$ of Circuit $C$ **do**
4:    C.add(g)
5:    **for** All inputs $g_j$ of $g_i$ **do**
6:       **if** $!(g_j \in L_{fo} \cup PI)$ **then**
7:          $C.add(g_j)$
8:          Call recursive for all inputs of $g_j$ over $Imp$
9:          Add found gates to $C$
10:      **end if**
11:   **end for**
12:   $Cones = Cones \cup C$
13: **end for**
14: **Return:** Cones
```

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 4: Bug Localization Algorithm

```
1: **Input:** Partitioned Netlist, Faulty Outputs $E$
2: **Output:** Suspected Regions $C_S$
3: **for** each faulty output $e_i \in E$ **do**
4:    find cones that construct $e_i$ and put in $C_{e_i}$
5: **end for**
6: $C_S = C_{e_0}$
7: **for** $e_i \in E$ **do**
8:    $C_S = C_S \cap C_{e_i}$
9: **end for**
10: **Return:** $C_S$
```

Algorithm 4 shows the bug localization procedure. Given a partitioned erroneous circuit and a set of faulty outputs E, the goal of the automatic bug localization is to identify all of the potentially responsible cones for the error. First, we find sets of cones $C_{e_i} = \{c_1, c_2, ..., c_j\}$ that constructs the value of each e_i from set E (line 4-5). These cones contain suspicious gates. We intersect all of the suspicious cones C_{e_i}s to prune the search space and improve the efficiency of bug 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 faulty behavior in just one of the outputs, we can conclude that the location of the bug is in the cone that generates this output. Otherwise, the location of the bug is in the intersection of cones which constructs the faulty outputs. We use this information to detect and correct the bug of the circuit. We describe the details of debugging in Section 4.1.3.

Example 3: Consider the faulty 2-bit multiplier shown in Fig. 4-5. 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 fault are calculated based on method demonstrated in Section 8-3. 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.

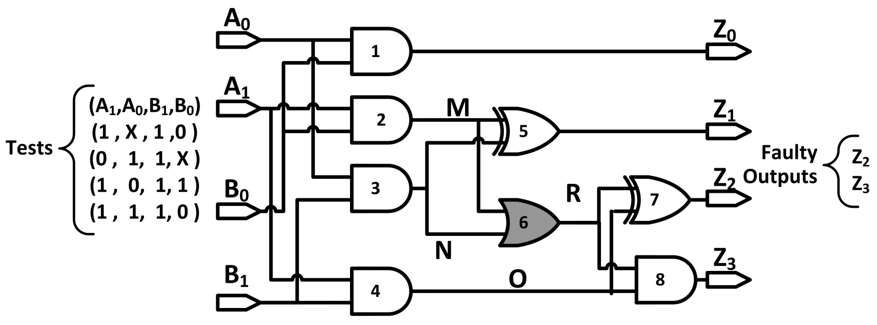


Figure 4-5. Faulty gate-level netlist of a 2-bit multiplier with associated tests

4.1.3 Error Detection and Correction

After test generation and bug localization, the next step is error detection. The remainder is helpful since it contains valuable information about the nature of the bug and its location. For example, when the faulty gate is located in the first level (inputs of faulty gates are primary inputs), it creates certain patterns in the remainder. These specific patterns are due to the termination of the substitution process in equivalence checking after this level, which prevents errors from propagating any further. In Example 1, the first level OR gate is placed by mistake instead of an AND gate. Let us consider the effect of the bug from algebraic point of view: the equivalent algebraic value of M is $M = A_1 + B_0 - A_1.B_0$ in the erroneous implementation; however, in the correct implementation, M should be equal to $M^* = A_1.B_0$. Thus, the difference between M and M^*, $(A_1 + B_0 - 2.A_1.B_0)$ with a coefficient will be observed in the remainder. Therefore, whenever $a+b-2.a.b$ pattern is seen in the remainder and there is an OR gate with inputs (a, b) in the implementation, we can conclude that the OR gate is the source of error and it should be replaced with an AND gate. Table 4-2 shows the patterns that will be observed for mis-placement of different types of gates. Note that, 3-input (or more) gates can be modeled as cascades of 2-input gates. So, the patterns are also valid for complex gates.

Table 4-2. Remainder patterns caused by gate misplacement error

Suspicious Gate	Appeared Remainder's Pattern	Solution
AND (a,b)	P_1 : -a-b+2.a.b	S_1 : OR (a,b)
	P_2 : -a-b+3.a.b	S_2 : XOR (a,b)
OR (a,b)	P_1 : a+b-2.a.b	S_1 : AND (a,b)
	P_2 : a.b	S_2 : XOR (a,b)
XOR (a,b)	P_1 : a+b-3.a.b	S_1 : AND (a,b)
	P_2 :-a.b	S_2 : OR (a,b)

From Section 4.1.2, we have a set of cones C_S such that their gates are potentially responsible for the bug. First, the gates in C_S are extracted and they are kept in a set G. Next, the suspicious gates in first level from G are considered and the remainder is scanned to check whether one of the patterns in Table 4-2 is recognized. If the pattern is found, the faulty gate is replaced with the corresponding gate. Otherwise, the terms of the remainder are rewritten such that it contains output variable of first level gates (at this time, we are sure that the first level gates are not the cause of the problem). We also remove the non-faulty gates from G. Then, we repeat the process over the remaining gates in G until we find the source of the error.

Example 4: Consider the faulty circuit shown in Example 3. The remainder is $R = 4.A_1.B_0 + 4.A_0.B_1 - 8.A_0.A_1.B_0.B_1$ and the potentially faulty gates are numbered $2, 3, 4, 6$. As we can see, remainder R does not contain any patterns shown in Table 4-2. It means that the first level suspicious gates 2, 3 and 4 are not responsible for the fault. Thus, we try to rewrite the remainder's terms with the output of the correct gates. In this step, we know that gates 2 , 3 and 4 are correct so their algebraic expressions are also true. As 6 is the only remaining gate, it is the answer. However, we continue the process to show the proof. By considering $M = A_1.B_0$ and $N = A_0.B_1$, R will be rewritten as $R^* = 4.(M + N - 2.M.N)$ (the GCD of coefficients of remainder terms is computed and the remainder is divided over GCD or signal's weight is computed as shown in [58]). Now, we consider the gates in the second level. This time R^* has one of the patterns shown in the Table 4-2. Based on Table 4-2, an AND gate with (M, N) as its inputs has been replaced with an OR gate. The only gate that has these characteristics is gate 6 which is also in G. It means that the source of the error has to be the gate 6 and if replaced with an AND gate, the bug will be corrected.

Finding and factorizing of remainder terms in order to rewrite them would be complex for larger designs. To overcome the complexity and obviate the need for manual intervention, we propose an automated approach shown in Algorithm 5. The algorithm takes faulty gate-level netlist, remainder R and potentially faulty gates of set G sorted based on their levels as inputs. It starts from first level gate g_i; if g_i is the buggy gate, one of the patterns in Table 4-2 should have been manifested in the remainder based on g_i's type. Therefore, the debugging algorithm computes two patterns (P_1, P_2) with g_i's inputs (lines 7-12) and scan the remainder to check whether one of them matches. If one of the patterns is found, the bug is identified and it can be corrected based on Table 4-2 (lines 13-16). Otherwise, g_i is correct and it will be removed from set G and next gate will be selected. Moreover, the current algebraic expression of g_i is true and it can be used in subsequent iterations (gate g_j from upper levels gets output of g_i as one of its inputs, the expression of g_i can be used instead of its output variables). As we want to compute patterns such that they contain just primary inputs (weight of gates' output is computed based on [58]), we use a dictionary to keep the expression of the gate output based on the primary inputs (line 19). The weight of each gates' output is computed from considering known weight of primary inputs and primary outputs and moving from backward and forward considering the fact that for XOR and OR gates, the output's weight is equal to inputs weight. In multipliers, the output's weight of the first level AND gates is computed as multiplication of inputs' weights (they are responsible for partial products). On the other hand, the output's weight of other AND gate in the design is computed as the summation of inputs' weights (since they are mostly used in half adders). In Adders, the output's weight of all AND gates is computed as summation input's weights. The process continues until the bug is detected or set G is empty. As, suspicious gates form a cone format, when the algorithm starts from primary inputs, it will not reach a gate whose inputs do not exist in the dictionary. Note that, our debugging approach does not need all of the counterexamples to work. It works even if there is no counterexample (all of the gates are considered as suspicious) or there is just one counterexample. However, having more counterexamples improves debug performance.

Example 5: We want to apply Algorithm 5 on the case shown in Example 4. We start from gate 2 and compute $P_1 = -A_1 - B_0 + 2.A_1.B_0$ and $P_2 = -A_1 - B_0 + 3.A_1.B_0$ for gate 2. As these patterns do not exist in the remainder, gate 2 is correct and the dictionary will be updated as $(M = A_1.B_0)$. The same will happen for gate 3 and 4 and dictionary will be updated as $(M = A_1.B_0, N = A_0.B_1)$ at the end of this iteration. Now, the gate 6 is considered and the P_is are as follows: $P_1 = A_1.B_0 + A_0.B_1 - 2.A_1.B_0.A_0.B_1$ and $P_2 = A_1.B_0.A_0.B_1$. Considering that $R = 4(A_1.B_0 + 4.A_0.B_1 - 2.A_0.A_1.B_0.B_1)$, P_2 of gate 6 can be observed in R. So the bug is the OR gate 6 and based on Table 4-2 it will be fixed by replacing with an AND gate.

Algorithm 5: Error Detection/Correction

```
1: Input: Suspected Gates $G$, Remainder $R$
2: Output: Faulty Gate and Solution
3: sort $g_i$ based on their levels (lowest level first)
4: for each level $j$ do
5:    for each $g_i \in G$ from level $j$ do
6:       $(a, b) = inputs(g_i)$
7:       if !( each of $(a, b)$ are from PI) then
8:          $a = dic.get(a)$
9:          $b = dic.get(b)$
10:      end if
11:      $P_1 = ComputeP_1(a, b)$
12:      $P_2 = ComputeP_2(a, b)$
13:      if ($P_1$ is found in $R$) then
14:         Return: gate $g_i$ and solution $S_1$ from Table 4-2
15:      end if
16:      if ($P_2$ is found in $R$) then
17:         Return: gate $g_i$ and solution $S_2$ from Table 4-2
18:      else
19:         remove $g_i$ from $G$
20:         dic.add(output($g_i$), Expression($g_i(a, b)$))
21:      end if
22:   end for
23: end for
```

4.2 Debugging Multiple Bugs

In this section, we want to extend the presented approach in Section 4.1 to be able to debug a faulty implementation of an arithmetic circuit with multiple bugs (we consider gate misplacement as our fault model). If the equivalence checking of an arithmetic circuit results to a non-zero remainder, we know that the implementation is buggy. However, the sources of the errors are unknown. Our plan is to use the non-zero remainder in order to generate directed tests to activate the bugs, localize the source of errors and correct them. First, we explain how we extend the approach presented in Section 4.1 to correct multiple independent bugs. Then, we present an approach to solve the debugging problem of an arithmetic circuit with two dependent misplaced gates. Figure 4-6 shows different steps of our proposed debugging approach to detect and correct multiple bugs.

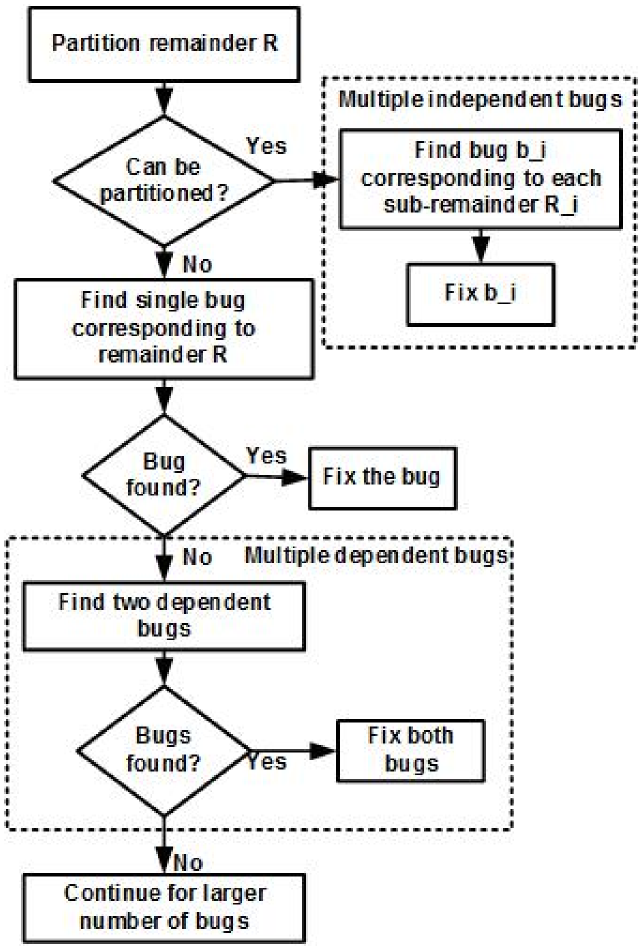

Partition remainder R
Can be partitioned?
Yes
No
Multiple independent bugs
Find bug b_i corresponding to each sub-remainder R_i
Fix b_i
Find single bug corresponding to remainder R
Bug found?
Yes
Fix the bug
No
Multiple dependent bugs
Find two dependent bugs
Bugs found?
Yes
Fix both bugs
No
Continue for larger number of bugs

Figure 4-6. Overview of different steps of our proposed debugging framework.

If there are more than one bug in the implementation, the remainder will be affected by all of them since all of the faulty gates are contributing in the equivalence checking procedure as well as the remainder generation. In other words, the remainder shows the effect of all (unknown) bugs exist in the implementation. Example 6 shows how the remainder is generated when there are two bugs in the implementation.

Example 6: In the circuit shown in Figure 4-7, the AND gate with inputs (A_0, B_0) as well as the AND gate with inputs (A_1, B_1) are replaced with XOR and OR gates by mistake respectively (there are two faults in the implementation of a 2-bit multiplier). The result of equivalence checking (remainder polynomial) can be computed as shown in Equation 4–2.

$$\begin{aligned}
&f_{spec} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&step_1 : 4.R + 4.O + 2.z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&step_2 : 4.O + 2.M + 2.N + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&step_3(remiander) : R = A_0 + B_0 - 3.A_0.B_0 + 4.A_1 + 4.B_1 - 8.A_1.B_1
\end{aligned} \tag{4–2}$$

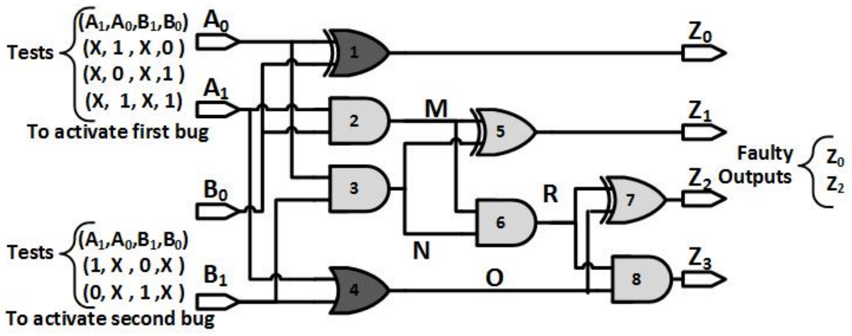


Figure 4-7. Gate-level netlist of a 2-bit multiplier with two bugs (dark gates) as well as associated tests to activate them.

Detailed observation in the remainder generation procedure shows that the overall remainder can be considered as a summation of different individual bug's effect in the equivalence checking process. For instance, one part of the remainder shown in Example 3, comes from the remainder shown in Example 1 (the same bug) as $(A_0 + B_0 - 3.A_0.B_0)$ and the other part $(4.A_1 + 4.B_1 - 8.A_1.B_1)$ is responsible for the second bug and it is equal to the remainder that can be the result of the equivalence checking with an implementation which contains only the second bug. Therefore, each assignment that makes the remainder non-zero activates at least one of the existing faulty scenarios. Some tests may activate all of the bugs at the same time. Thus, Algorithm 2 can be used to generate directed tests when there are more than one fault in the design.

Example 7: Directed test to activate the buggy implementation of Example 6 are shown in Figure 7. The assignments make the first part of the remainder non-zero $(A_0 + B_0 - 3.A_0.B_0)$, activates the first fault. For example, assignment $(A_1 = 1, A_0 = 0, B_1 = 0, B_0 = 0)$ manifests the effect of the first fault in Z_0. On the other hand, the assignments that make the second part of the remainder non-zero $(4.A_1 + 4.B_1 - 8.A_1.B_1)$, are tests to activate the second bugs. Assignment ($A_1 = 1, A_0 = 0, B_1 = 0, B_0 = 0$) activates the second fault in Z_2. However, the assignment ($A_1 = 1, A_0 = 0, B_1 = 0, B_0 = 1$) activates both of the faults at the same time (Z_0 and Z_2).

To localize the source of errors, the generated tests are simulated to find faulty primary outputs. Faulty gates exist in the cones that construct the functionality of faulty outputs. In order to prune the search space and localize source of errors, we cannot directly apply Algorithm 4 as their intersection may be a zero set. However, some information can be found from using Algorithm 4. In following sections, we explained the bug localization, bug detection, and correction for multiple bugs correction in two different scenarios: i) bugs with independent input cones, ii) bugs which share some input cones.

4.2.1 Error Correction for Multiple Independent Bugs

We call two bugs independent of each other if they have different input cones (fan-ins). Figure 4-7 shows two independent in a 2-bit multiplier. If multiple bugs are independent of each other, their effect can be observed easily in the remainder as a summation of each individual bug's remainder (summation of sub-remainders). Therefore, if the remainder is partitioned into multiple sub-remainders based on the primary inputs (each part representing the effect of one bug), each sub-remainder as well as the associate faulty cones can be fed into Algorithm 5 in order to detect and correct the source of each independent errors.

If the input cones (input fan-ins) of faulty gates are separate from each other, a different set of primary inputs may appear in each sub-remainders. In order to find the sub-remainders, each term of the overall remainder and its corresponding monomial is examined to determine which sub-remainder it belongs. Algorithm 6 shows the remainder partitioning procedure.

Algorithm 6 takes the overall remainder R as input and returns the containing sub-remainders R_is. The algorithm sorts the terms of the R based on their monomial size (the number of variables exists in each term) in descending order (line 5). In the next

Algorithm 6: Remainder Partitioning

```
1: **Input:** Remainder $R$
2: **Output:** Sub-remainders $\mathbb{R}$
3: Input: Remainder R
4: Output: Sub-remainders $\mathbb{R}$
5: Sort terms of $R$ based on their size
6: $R_0 = largestTerm(R)$
7: $\mathbb{R} = \{R_0\}$
8: for each term $t \in R$ do
9:     for each sub-remainder $R_i \subset \mathbb{R}$ do
10:        if ($R_i$ contains some of the variable $t$) then
11:            $R_i = R_i + t$
12:        else
13:            new $R_j = t$
14:            $\mathbb{R} = \mathbb{R} \cup R_j$
15:        end if
16:    end for
17: end for
18: Return: $\mathbb{R}$
```

step, it starts from the largest term of the remainder R and adds it to sub-remainder R_0 (line 6). Then, it examines all terms of R from the second largest term t to find out which partition they belong (lines 7-8). If some of the variables which exist in the t already exist in terms of sub-remainder R_i, term t will be added to sub-remainder R_i (lines 9-10). Otherwise, the algorithm creates a new sub-remainder R_j and adds t to it (lines 12-13). The process continues until all terms of the R are examined. If the algorithm results to one sub-remainder, it shows that faulty gates do not have independent input cones. The computed sub-remainders are fed into Algorithm 2 in order to generate directed tests activating the corresponding bug of that sub-remainder. The generated test are used to define the corresponding faulty outputs of each bug. Example 8 illustrates the remainder partitioning procedure.

Example 8: Consider the faulty multiplier design shown in Figure 4-7 and corresponding remainder shown in Equation 4–2. To be able to find different possible sub-remainders, the remainder is sorted as: $R = -3.A_0.B_0 - 8.A_1.B_1 + A_0 + B_0 + 4.A_1 + 4.B_1$. The partitioning starts from term $-3.A_0.B_0$ and as there are no sub-remainder so far, sub remainder R_1 is created and the term will be added to it as: $R_1 = -3.A_0.B_0$. The second term $-8.A_1.B_1$ is examined and as R_1 does not contains variables A_1 and B_1, new sub-remainder R_2 is created. Similarly, rest of the terms of R are examined and R_1 and R_2 are computed as: $R_1 = -3.A_0.B_0 + A_0 + B_0$ and $R_2 = -8.A_1.B_1 + 4.A_1 + 4.B_1$. Corresponding tests of each sub-remainder are shown in Figure 4-7.

Corresponding Tests of each sub-remainder are simulated and faulty outputs are defined. The faulty outputs of each bug are fed into Algorithm 4 in order to find potential faulty cones. Algorithm 5 is used with each sub-remainder as well as corresponding potential faulty gates as its inputs and it tries to detect and correct of each bug. In other words, the problem of debugging a faulty design with n independent bugs is mapped to debugging of n faulty designs where each design contains a single bug. We illustrate how to apply Algorithm 5 to correct multiple independent sources of error using Example 9.

Example 9: Having the directed tests shown in Figure 4-7, faulty outputs Z_0 and Z_2 as well as two sub-remainders computed in Example 7, Algorithm 5 is used twice to find the source of errors. In the first attempt, the faulty output is Z_0 and the computed potential faulty cone using Algorithm 4 contains gate 1. Therefore, gate 1 as well as R_1, are fed into the bug correction algorithm. Two patterns $P_1 = A_0 + B_0 - 3.A_0.B_0$ (if the potential faulty gate 1 should be an AND gate) and $P_2 = -1.A_0.B_0$ (if the potential faulty gate 1 should be an OR gate) are computed. Therefore, Gate 1 should be replaced with an AND gate to fix the first bug since the P_1 is equal to the remainder. The same procedure happens for the second bugs while the potential faulty gates are $\{2, 3, 4, 6, 7\}$ since the only faulty output is Z_2. Trying different patterns results in a conclusion that gate 4 should be replaced with an AND gate.

4.2.2 Error Correction for Two Bugs with Common Input Cones

In this section, we describe how to detect and correct two bugs that they do not have independent input cones. The key difference here from the cases that we solved in Section 4.2.1 is the fact that the remainder cannot easily be partitioned into sub-remainders since some of the terms of the corresponding sub-remainder may be canceled through other sub-remainders or they may be combined to each other. The reason is that the bugs share some input cones (fan-ins) individual sub-remainders may have common terms where contain a certain set of primary inputs as variables. When sub-remainders are combined to each other to form the overall remainder, some term combinations/cancellations happen. Moreover, some of the sub-remainders may be affected by lower level faults and the presented method in Section 4.2.1 cannot solve these cases. We illustrate the fact using the following example.

Example 10: Consider the faulty implementation of a 2-bit multiplier with two bugs as shown in Figure 4-8. Gates 6 and 7 are misplaced with OR gates. it can be observed from Figure 4-8 that two bugs share some set of input cones (gates $\{2, 3, 4\}$ are common in input cones of faulty gates 6 and 7). Applying equivalence checking on the circuit shown in Figure 4-8 results in a non-zero remainder: $R = 8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$. However, if only gate 6 is misplaced with an OR gate in the implementation (single bug), the remainder will be equal to: $R_1 = 4.A_0.B_1 + 4.A_1.B_0 - 8.A_0.A_1.B_0.B_1$. Similarly, when only gate 7 is misplaced with an OR gate (single fault), the remainder will be computed as: $R_2 = 8.0.A_1.B_1 - 8.0.A_0.A_1.B_0.B_1$. As it can be observed, $R \neq R_1 + R_2$. The reason is that buggy gate 6 has an effect on the generation of sub-remainder R_2. As a result, $R\prime_2$ should be computed as: $R\prime_2 = 8.0.A_1.B_1 + 8.0.A_0.B_1 + 8.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 + 8.0.A_0.A_1.B_0.B_1$. Now, it can be seen that the $R = R_1 + R\prime_2$. Note that there is not any monomial of $A_0.A_1.B_0.B_1$ in the remainder R; however, this monomial exists in both R_1 and $R\prime_2$ with opposite coefficients (term cancellation happens).

As it can be observed from Example 10, term cancellation as well as lower level bugs' effect are two main reasons that limit the applicability of the presented method in Section 4.2.1 to detect and correct bugs with common input cones. In this section, we

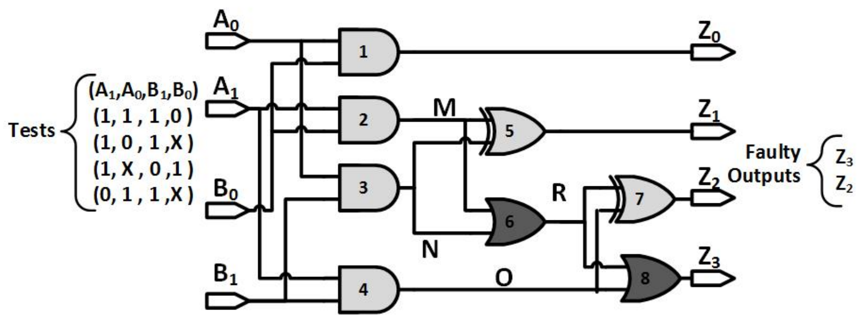


Figure 4-8. Gate-level netlist of a 2-bit multiplier with two bugs (dark gates) which shares some input cones as well as associated tests to activate them.

present a general approach to correct and detect two gate misplacement bugs regardless of the bugs' positions.

The first step to fix unknown dependent bugs is to use of Algorithm 2 in order to generate directed tests to activate unknown bugs. In the next step, tests are simulated to define the faulty outputs (E) which the effect of faults can be propagated to them. Algorithm 4 cannot be used to localized the potential faulty cones since the intersection of faulty cones may eliminate some of faulty gates. Instead, union of all of the gates that construct faulty outputs should be considered as potential faulty gate candidates to make sure that all of the potential faulty gates are considered. The next step is to define faulty gates and offer their solutions by using the remainder as well as potential faulty gates.

We are looking for two sub-remainders that their summation constructs the overall remainder R. Note that sub-remainder of an individual bug may be affected by the other existing bug in the implementation (for instance, sub-remainder $R\prime_2$ which shows the effect of faulty gate 7 in Example 10 contains the effect of faulty gate 6). Algorithm 7 is used to detect and correct two dependent bugs by finding two sub-remainders R_1 and R_2 where their summation is equal to R ($R = R_1 + R_2$). Therefore, it tries to find two equal polynomials: $R - R1$ and $R2$. The algorithm takes the remainder and potential faulty gates as inputs and it returns two faulty gates and their solutions as output. The algorithm contains two major steps: first, it constructs two patterns for each potentially faulty gates based on Table 4-2 regarding the functionality of their input gates (lines

7-10). For each pattern it computes the $R - P_i$ and it stores the result in a dictionary (lines 11-12). In the next step, each of the patterns P_j is checked whether it exists in the dictionary (line 15). If such pair exist in the remainder, $R_1 = P_i$ and $R_2 = P_j$ are two sub-remainders that we are looking for and their corresponding gates are faulty gates and their solution can be found based on Table 4-2 (lines 16-18). Note that, by using hash map $\mathbb{R}$ the complexity of the algorithm is proportional to number of faulty gates.

Algorithm 7: Debugging Two Dependent Bugs

```
1:  **Input:** Suspicious gates $\mathbb{G}$, remainder $R$
2:  **Output:** Faulty gates and their solution
3:  $\mathbb{P} = \{\}$ {keeps patterns for all gates as well as corresponding solution of each pattern}
4:  $\mathbb{R} = \{\}$ {keeps remainder minus all patterns}
5:  **for** each gate $g \in \mathbb{G}$ **do**
6:    $(a, b)$=getInputPolynomials($g$)
7:    $P_1 = computeP1(a, b)$
8:    $P_2 = computeP2(a, b)$
9:    $\mathbb{P} = \mathbb{P} \cup \{P_1, P_2\}$
10:   $\mathbb{R}.put((R - P_1), P_1)$
11:   $\mathbb{R}.put((R - P_2), P_2)$
12: **end for**
13: **for** each $P_j \in \mathbb{P}$ **do**
14:   **if** $P_j$ exists in $\mathbb{R}$ **then**
15:     $P_i = \mathbb{R}.get(P_j)$
16:     gate $g_i = \mathbb{P}.get(P_i)$ is faulty and get solution $S_i$ from Table 4-2
17:     gate $g_j = \mathbb{P}.get(P_j)$ is faulty and get solution $S_j$ from Table 4-2
18:   **end if**
19: **end for**
```

Example 11: Consider the faulty implementation of a 2-bit multiplier shown in Figure 4-8 with remainder: $R = 8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$. Corresponding directed tests to activate existing bugs and faulty gates are shown in Figure 4-8. Potential faulty gates are computed based on faulty outputs Z_2 and Z_3 as gates $\{2, 3, 4, 6, 7, 8\}$. Patterns, their possible solution as well remainder minus patterns is listed in Table 4-3. Note that, Table 4-3 is the combination of two list $\mathbb{P}$ and hash map $\mathbb{R}$ which are mentioned in Algorithm 7. Each pattern listed in the second column is tested whether exists in hash map $\mathbb{R}$ (part of hash map is shown in the fourth column). As it can

Table 4-3. Patterns for potential faulty gates Example 11

Gate#	Pattern	solution	Remainder minus pattern
2	$2.A_1 + 2.B_0 - 4.A_1.B_0$	OR	$-2.A_1 - 2.B_0 + 8.A_1.B_1 + 16.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
	$2.A_1 + 2.B_0 - 6.A_1.B_0$	XOR	$-2.A_1 - 2.B_0 + 8.A_1.B_1 + 18.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
3	$2.A_0 + 2.B_1 - 4.A_0.B_1$	OR	$-2.A_0 - 2.B_1 + 8.A_1.B_1 + 12.A_1.B_0 + 16.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
	$2.A_0 + 2.B_1 - 6.A_0.B_1$	XOR	$-2.A_0 - 2.B_1 + 8.A_1.B_1 + 12.A_1.B_0 + 18.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
4	$4.A_1 + 4.B_1 - 8.A_1.B_1$	OR	$-4.A_1 - 4.B_1 + 16.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
	$4.A_1 + 4.B_1 - 12.A_1.B_1$	XOR	$-4.A_1 - 4.B_1 + 20.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
6	$4.A_0.B_1 + 4.A_1.B_0 - 8.A_0.A_1.B_0.B_1$	AND	$8.A_1.B_1 + 8.A_1.B_0 + 8.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 + 8.A_0.A_1.B_0.B_1$
	$4.A_0.A_1.B_0.B_1$	XOR	$8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 + 4.A_0.A_1.B_0.B_1$
7	$4.A_0.B_1 + 4.A_1.B_0 + 4.A_1.B_1 - 8.A_0.A_1.B_0 - 8.A_1.B_0.B_1 + 4.A_0.A_1.B_0.B_1$	AND	$4.A_1.B_1 + 8.A_1.B_0 + 8.A_0.B_1 - 8.A_0.A_1.B_0 - 8.A_1.B_0.B_1 + 8.A_0.A_1.B_0.B_1$
	$4.A_0.A_1.B_0 + 4.A_1.B_0.B_1 - 4.A_0.A_1.B_0.B_1$	OR	$8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 20.A_0.A_1.B_0 - 20.A_1.B_0.B_1 + 4.A_0.A_1.B_0.B_1$
8	$8.0.A_1.B_1 + 8.0.A_0.B_1 + 8.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1 + 8.0.A_0.A_1.B_0.B_1$	AND	$8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 16.A_0.A_1.B_0 - 16.A_1.B_0.B_1$
	$8.A_0.A_1.B_0 + 8.A_1.B_0.B_1 - 8.A_0.A_1.B_0.B_1$	XOR	$8.A_1.B_1 + 12.A_1.B_0 + 12.A_0.B_1 - 24.A_0.A_1.B_0 - 24.A_1.B_0.B_1 + 8.A_0.A_1.B_0.B_1$

be seen in the table, the fourth column contains the highlighted polynomial of the second column. As it can be seen, the highlighted polynomials are equal. It means that gate 6 and 7 are faulty and they should be substituted with AND gates.

If the algorithm could not find two bugs responsible for the errors, the algorithm should be continued for a larger number of bugs (for example, finding three sub-remainders that their summation construct the remainder). Note that, Algorithm 7 requires to construct the exact sub-remainder responsible for the potential bugs (it is not useful to find the pattern as some part of the remainder). In arithmetic circuit implementations, most of the gates are connected to half-adders or they are in the last level of the design. Therefore, if we consider them as potentially faulty gates, their constructed patterns are equal to the exact remainder. However, if they are not in the last level of the design and they are not connected to a half-adder, the exact sub-remainder is also dependent on the structure of the next level. To illustrate the point, suppose that we assume that gate g_1 with functionality f_{g_1} may be misplaced by polynomial $f_{g_1'}$. Then the constructed pattern is computed as: $\Delta = f_{g_1'} - f_{g_1}$. If gate g_1 is connected to a half-adder with inputs g_1 and g_2, the exact sub-remainder is computed as: $\Delta - 2.\Delta.f_{g_2} + 2.\Delta.f_{g_2} = \Delta$. However, if it is only connected to a XOR gate g_2, the exact sub-remainder would be equal to: $\Delta - 2.\Delta.f_{g_2}$. Note that, if we have two cascaded bugs, this effect only may happen for the higher level bug since the effect of the lower level bug is considered while constructing the pattern of the higher level bug.

4.3 Experiments

4.3.1 Experimental Setup

The directed test generation, bug localization, and bug detection algorithms were implemented in a Java program and experiments were conducted on a Windows PC with Intel Xeon Processor and 16 GB memory. We have tested our approach on both pre- [93] and post-synthesized gate-level arithmetic circuits that implement adders and multipliers. Post-synthesized designs were obtained by synthesizing the high-level description of arithmetic circuits using Xilinx synthesis tool. We consider gate misplacement or signal inversion which change the functionality of the design as our fault model. Several gates from different levels were replaced with an erroneous gate in order to generate faulty implementations. The remainders were generated based on the method presented in [93]. Multiple counterexamples (directed tests) are generated based on one remainder. As each counterexample can be generated independent of others, so we used a parallelized version of the algorithm for faster test generation. We compared our test generation method with existing directed test generation method [33] as well as random test generation. To start our debugging procedure, we use the generated counterexamples in test generation phase and find faulty primary outputs. Then, we run the bug localization algorithm that takes faulty outputs as input. In the next step, we apply our debugging algorithm on suspicious areas that bug localizer has identified. Note that, our debugging procedure does not need the suspicious gates to work and in the worst case, it considers all of the gates suspicious. However, using bug localization algorithm improves our method drastically. As remainder may explode when bugs are closer to the primary outputs, these bugs are harder to detect especially when the size of the circuit is very large. On the other hand, the bugs that are closer to the primary inputs are easier to detect. We have inserted several bugs in the middle levels of the circuits to conduct our experimental results. We compared our debugging results with most recent work in this context and we use the benchmarks obtained from the authors [58]. However, we have implemented their algorithm to be able

Table 4-4. Results for Debugging One Error in Arithmetic Circuits. TO = timeout after 3600 sec; MO = memory out of 8 GB

Benchmark			Test Generation (TG)			Bug Localization (BL)	Debugging/Correction (DC)		
Type	Size	# Gates	[33] (s)	Random(s)	Our TG(s)	Bug Loc.(s)	[58]	Our (TG+BL+DC)	Improvement
post-syn. Multipliers	4	72	1.88	0.02	0.01	0.001	0.2	0.02	10x
	16	1632	42.69	1.48	0.32	0.03	4.32	0.78	5.5x
	32	6848	205.66	3.03	0.82	0.16	18.50	2.40	10.94x
	64	28K	MO	16.97	1.65	0.83	151.05	13.63	11.08x
	128	132K	MO	66.52	3.83	5.1	1796.50	52.91	33.95x
	256	640K	MO	TO	15.65	22.39	TO	205.01	-
pre-syn. Multipliers	4	94	1.27	0.04	0.01	0.001	0.17	0.03	5.6x
	16	1860	43.11	1.93	0.4	0.03	4.45	0.83	5.36x
	32	7812	189.50	5.69	0.87	0.2	23.1	2.67	8.65x
	64	32K	MO	29.07	1.77	0.8	180.3	14.91	12.09x
	128	129K	MO	83.60	4.1	3.8	1743.07	47.74	36.51x
	256	521K	MO	TO	12.44	15.83	TO	170.48	-
post-syn. Adder	64	573	154.97	1.51	0.5	0.01	3.12	0.71	4.39x
	128	1251	MO	3.48	1.07	0.05	6.60	1.69	3.90x
	256	2301	MO	10.64	3.09	0.05	17.32	4.27	3.35x
pre-syn. Adder	64	444	128.12	1.15	0.35	0.01	2.95	0.51	5.78x
	128	880	MO	4.40	0.84	0.03	6.46	1.12	5.76x
	256	1698	MO	9.10	2.23	0.1	16.18	3.54	2.05x

to compare our method with their method. To enable fair comparison, similar to [58], we randomly inserted bugs (gate changes) in the middle stages of the circuits. We improved the run-time complexity of presented method in [52] by using efficient data structures such as hash maps and sorted sets.

4.3.2 Debugging a Single Error

Table 4-4 presents results for test generation, bug localization and debugging methods using multipliers and adders. The first column indicates the types of benchmarks. The second and third columns show the size of operands and number of gates in each design, respectively. Since the sizes of adder designs are smaller than multiplier designs, we show results only for higher operand sizes (bit-widths). The fourth column indicates results for directed test generation method presented in [33] by using SMV model checker [28] (We give the model checker the advantage of knowing the bug). The fifth column represents results of random test generation method (time to generate the first counterexample using the random technique). The sixth column represents the time of our test generation method that generates multiple tests. As it can be observed from Table 4-4, our method has improved directed test generation time by several orders of magnitude. The seventh column shows the CPU time for bug localization algorithm. The eighth column shows the debugging time of [58] using our implementation in Java. The next column provides CPU time of our proposed approach which is the summation of test generation (TG), bug localization (BL) and debugging/correction (DC) time. The last column shows the improvement provided by our debugging framework. Clearly, our approach is an order-of-magnitude faster than the most closely related approach [58], especially for larger designs as bug localization has an important effect. The reported numbers are the average of generated results for several different scenarios. For instance, if we zoom in test generation of the first row (post-synthesized multiplier with 4-bit operands) of Table 4-4, the reported results are the average of the nine possible scenarios shown in Table 4-5.

Table 4-5. Test Generation for 4-bit multiplier with 8-bit outputs

Faults	[33]	Ran. tests(ms)	#tests	Faulty outputs	# Ran.	Our TG(s)
$XOR \rightarrow AND$	1.48	47.70	18	Z_7, Z_6, Z_5, Z_4	2632	0.01
$XOR \rightarrow OR$	2.12	25.95	4	Z_2	2945	0.01
$XOR \rightarrow AND$	1.95	19.21	128	Z_4	2292	0.01
$XOR \rightarrow OR$	2.27	26.43	12	Z_6, Z_5, Z_4, Z_3	2945	0.05
$XOR \rightarrow AND$	1.03	16.31	14	Z_6, Z_5, Z_4, Z_3, Z_2	2369	0.02
$AND \rightarrow XOR$	2.44	0.47	3	Z_6, Z_5, Z_4, Z_3, Z_2	1881	0.01
$AND \rightarrow OR$	2.20	1.90	2	Z_7, Z_6, Z_5	2258	0.01
$AND \rightarrow XOR$	0.89	44.17	148	Z_7, Z_6, Z_5, Z_4	2164	0.03
$OR \rightarrow AND$	2.52	11.51	148	Z_6	2920	0.01
Average	1.88	21.52	53	-	2489.55	0.01

Table 4-5 presents the debugging results of 4-bit post-synthesized multiplier. The first column shows a possible set of gate misplacement faults. Time to generate the first counterexample using [33] and random techniques are reported in second and third columns, respectively. The fourth column shows the number of directed tests generated by our approach to activate the bug (each of them activates the bug). The fifth column lists the outputs that are affected by the fault (activated by the respective tests reported in the fourth column). The sixth column shows the number of random tests required to cover all of our directed tests. It demonstrates that even for such small circuits, using random tests to activate the error is impractical. The last column shows our test generation time. As mentioned earlier, the average of these scenarios is reported in the first row of Table 4-5.

The experimental results demonstrated three important aspects of our approach. First, our test generation method generates multiple directed tests when the bug is unknown in a cost-effective way. Second, our debugging approach detects and corrects single fault caused by gate misplacement in a reasonable time. Finally, our debugging method is not dependent on any specific architecture of arithmetic circuits and it can be applied on both pre-synthesized and post-synthesized gate-level circuits.

4.3.3 Debugging Multiple Errors

Table 4-6 presents results for remainder-partitioning, test generation, bug localization and debugging methods using multipliers and adders with multiple independent bugs. The first column indicates the types of benchmarks. The second and third columns show the size of operands and number of bugs in each design, respectively. The fourth column represents the required time for remainder partitioning, and the fifth column represents the time of our test generation method. The sixth column shows the CPU time for bug localization algorithm. The seventh column shows the debugging time to detect and correct all bugs. The next column provides CPU time of our proposed approach which is the summation of remainder partitioning (RP), test generation (TG), bug localization (BL) and bug detection algorithm (DC) times. The ninth column shows the required time of method presented in [58] using our implementation in Java. The last column shows the improvement provided by our debugging framework. Clearly, our approach is an order-of-magnitude faster than the most closely related approach, especially for larger multipliers as bug localization has an important effect. However, our performance is comparable with [58] for debugging adders since the number of gates is small and the number of inputs is large and test generation time may surpass the speed up of our debugging method.

If remainder cannot be partitioned and Algorithm 5 cannot find a single source of error, we know that there are dependent bugs in the implementation. Therefore, Algorithm 7 is used to find dependent bugs. In this step, suspicious cones, as well as

Table 4-6. Debugging time of multipliers for multiple independent bugs. TO = timeout after 7200 sec.

Type	Size	#Bugs	RP(s)	TG(s)	Bug Loc.(s)	DC(s)	total (RP+TG+BL+DC)(s)	[58]	Improvement
post_syn. Multipliers	8x8	4	0.001	0.04	0.03	0.57	0.64	1.7	2.65x
		8	0.001	0.07	0.03	0.86	0.97	2.5	2.57x
	16x16	4	0.003	0.77	0.01	1.82	2.60	6.03	2.31x
		8	0.003	1.2	0.02	2.62	3.84	10.07	2.70x
	32x32	4	0.003	1.86	0.64	5.02	7.52	26.37	3.49 x
		8	0.003	2.08	1.18	9.4	12.67	43.98	3.47x
	64x64	4	0.006	5.65	3.9	38.48	48.03	178.89	3.72x
		8	0.006	7.06	4.7	65.31	78.07	250.07	3.206x
	128x128	4	0.008	11.59	10.1	124.52	146.22	1946.1	13.30x
		8	0.008	25.67	20.87	235.88	282.43	2337.56	8.28x
	256x256	4	0.012	39.58	70.65	508.42	618.66	TO	-
		8	0.012	65.21	122.01	906.22	1093.45	TO	-
pre_syn. Multipliers	8x8	4	0.001	0.44	0.03	0.35	0.82	1.73	2.11x
		8	0.001	0.5	0.03	0.66	1.19	2.67	2.24x
	16x16	4	0.002	1.3	0.05	2	3.35	7.4	2.21x
		8	0.002	1.90	0.05	2.87	4.79	10.05	2.1x
	32x32	4	0.003	2.08	0.73	5.8	8.61	30.34	3.52x
		8	0.003	3.23	1.31	11.98	16.52	43.18	2.61
	64x64	4	0.001	5.94	4.5	38.22	48.66	194	3.99x
		8	0.005	7.91	8.9	84.7	101.52	225.85	2.22x
	128x128	4	0.006	13.5	15.09	170.46	199.05	2036.37	10.36x
		8	0.006	22.48	26.72	207.88	257.09	2260.6	8.79x
	256x256	4	0.01	26.75	39.16	653	718.92	TO	-
		8	0.01	59.13	77.34	866.18	1002.66	TO	-
post_syn. Adders	64x64	4	0.004	1.09	0.07	0.37	1.53	3.43	2.24x
		8	0.003	2.63	0.12	0.47	3.23	3.85	1.19x
	128x128	4	0.005	3.34	0.2	0.71	4.25	7.59	1.78x
		8	0.01	5.41	0.37	1.25	7.24	8.72	1.2x
	256x256	4	0.01	8	0.3	5.62	13.93	19.87	1.42x
		8	0.01	13.44	0.8	9.94	24.19	25.93	1.07x
pre_syn. Adders	64x64	4	0.002	1.08	0.08	0.3	1.56	3.36	2.15x
		8	0.006	2.12	0.08	0.42	2.63	3.56	1.35x
	128x128	4	0.008	3.39	0.2	0.79	4.39	7.26	1.65x
		8	0.009	5.57	0.42	1.7	7.69	8.31	1.04x
	256x256	4	0.01	6.24	0.28	5.38	11.91	19.13	1.61x
		8	0.01	11.52	0.81	8.27	20.61	23.35	1.13x

patterns, are generated for single bug detection phase will be reused. We consider all of the gates in the suspicious cones as potential faulty gates (instead of intersecting faulty cones) to make sure that all sources of errors can be found. Algorithm 7 uses hash maps that map the string versions of polynomials to the possible solutions to able to perform a faster lookup and achieve a linear computation complexity proportional with the number of suspicious gates.

Table 4-7 presents results for remainder-partitioning, test generation, bug localization and debugging methods using multipliers and adders with two dependent bugs. The first column indicates the types of benchmarks. The second column shows the size of operands. The third column represents the required time for remainder partitioning, and the fourth column represents the time of our test generation method. The fifth and sixth columns show the CPU time for bug localization and debugging time, respectively. Bug localization time is relatively small in comparison with other scenarios since the intersection of faulty cones are not computed. The next column provides CPU time of our proposed approach which is the summation of remainder partitioning (RP), test generation (TG), bug localization (BL) and bug detection algorithm (DC) times. As the result shows, our approach can detect and correct multiple dependent bugs in reasonable time. We did not compare with any approaches since there are no existing approaches for detecting/fixing multiple dependent bugs.

Table 4-7. Debugging time of multipliers for two dependent bugs.

Type	Size	RP(s)	TG(s)	BL(s)	DC(s)	total (s)
post_syn. Mul.	8	0.001	0.1	0.01	0.98	1.09
	16	0.002	0.35	0.02	2.23	2.61
	32	0.002	0.96	0.08	13.92	14.94
	64	0.004	3.77	0.2	77.12	81.1
	128	0.008	8.06	0.6	241.05	249.71
	256	0.012	31.8	36.02	1099.96	1167.79
pre_syn. Mul.	8	0.001	0.1	0.01	0.91	1.02
	16	0.001	0.77	0.01	5	5.78
	32	0.002	1.03	0.08	13.54	14.65
	64	0.003	4.65	0.1	96.3	101.05
	128	0.005	7.88	0.6	220.22	228.70
	256	0.01	19.41	22.05	982.9	1024.37
post_syn. Mul.	64	0.001	01.18	0.01	0.55	1.74
	128	0.011	5.4	0.02	3.47	8.90
	256	0.011	16.09	0.1	9.42	25.62
pre_syn. Add.	64	0.003	1.13	0.01	0.53	1.67
	128	0.008	6.3	0.01	2.36	8.68
	256	0.01	10.97	0.08	15.04	24.10

4.4 Summary

In this chapter, we presented an automated methodology for debugging arithmetic circuits. Our methodology consists of efficient directed test generation, bug localization, and bug correction algorithms. We used the remainder produced by equivalence checking methods to generate directed tests that are guaranteed to activate the source of the bug when the bug is unknown. We used the generated tests to localize the source of the bug and find suspicious areas in the design. We also developed an efficient debugging algorithm that uses the remainder as well as suspicious areas to detect and correct the bug without any manual intervention. We extended the proposed approach to automatically detect and correct multiple bugs. Our experimental results demonstrated the effectiveness of our approach to solve debugging problem for large and complex arithmetic circuits by improving debug performance by an order-of-magnitude compared to the state-of-the-art approaches.

CHAPTER 5
INCREMENTAL ANOMALY DETECTION

Symbolic algebra is a promising approach to verify large and complex arithmetic circuits. Existing algebraic-based verification methods generate a remainder to indicate buggy implementation. The remainder is beneficial for debugging of the faulty implementation since it can be used for automated test generation, bug localization, and bug correction.

However, the effectiveness of existing equivalence checking approaches [52, 58] in fixing of the unknown Trojans is dependent on the result (generated remainder) of the equivalence checking techniques. None of the existing techniques [34, 121, 128] are capable of generating a remainder when the bug is deep inside the design. These methods are not scalable and leads to explosion in size of the remainder when the design is faulty. To make the matters worse, the location of the bug can also lead to the explosion in the number of remainder terms. In this chapter, we propose an incremental equivalence checking method to address the scalability challenges by solving the verification problem in the increasing order of design's input complexity. Our proposed approach makes two important contributions. It is able to generate smaller and compact remainders for large designs. Our proposed incremental debugging is capable of localizing and correcting multiple hard-to-detect bugs irrespective of their location in the design. Experimental results demonstrate that our approach can efficiently debug most difficult bugs in large arithmetic circuits when the state-of-the-art methods fail.

In this chapter, we address the above challenges by proposing an incremental debugging framework. The proposed approach partitions the primary inputs' space of the design in order to solve the verification and debug problems in the increasing order of the design complexity. The verification is performed in several iterations where the equivalence of the specification and implementation is checked with considering primary inputs' constraints. The basic intuition behind our work is to observe the fact that it is efficient to debug an error in a smaller region (e.g., the portion of the design that multiples

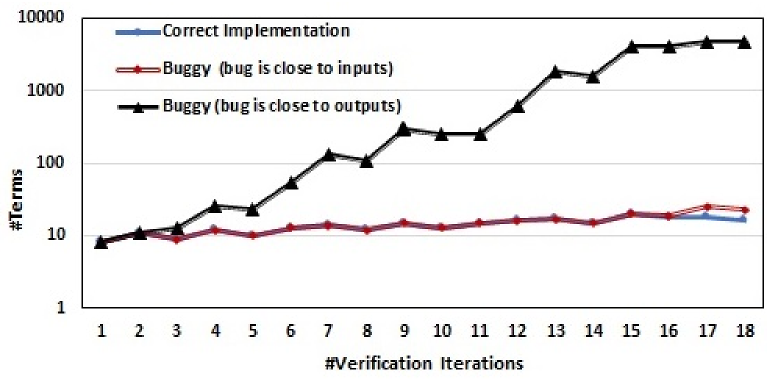


Figure 5-1. Comparison of the number of terms in different iterations in verification of a 4x4 multiplier when: i) the implementation is correct, ii) the implementation is buggy and the bug is close to the primary inputs, and iii) the implementation is buggy implementation and the bug is in the deeper stages of the design (e.g., close to the primary outputs).

the first two bits) instead of searching in the whole 4x4 multiplier. If no bugs found in the region representing 1×1 multiplication, in the next iteration a larger region (e.g., representing 2×2 multiplication) will be searched. On the surface, it may seem that our approach will take longer than solving directly on the original design, but as proposed work (Section 5.1) and results (Section 5.2) demonstrate that our well-crafted incremental approach drastically reduces the debugging complexity.

The rest of the chapter is organized as follows. Section 5.1 describes our proposed incremental equivalence checking and debugging framework. Section 5.2 presents our experimental results. Section 5.3 concludes the chapter.

5.1 Efficient Debugging of Arithmetic Circuits

In this chapter, we present an approach to incrementally perform equivalence checking between an arithmetic circuit specification and its implementation. *We consider gate misplacement that changes the functionality of the design as our fault model.* Figure 5-2 shows an overview of our proposed approach. Figure 5-2 highlights three key parts of our approach: i) finding an efficient order of primary inputs and partition the inputs' space into different constraints based on the given order; ii) incremental equivalence checking;

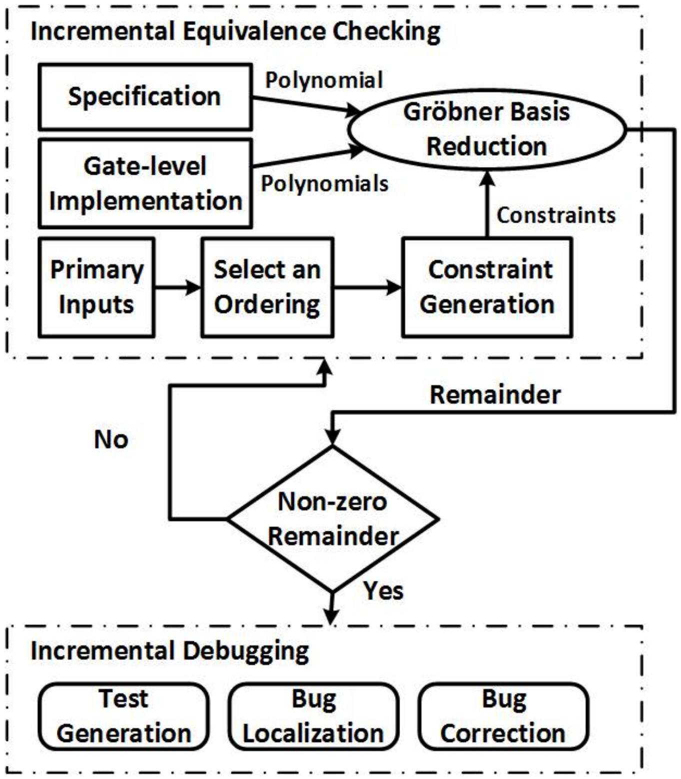


Figure 5-2. Overview of our proposed approach.

and iii) incremental debugging approach. The rest of this section describes these steps in detail.

5.1.1 Incremental Equivalence Checking

In this section, we present an approach to solve the equivalence checking problem in complex arithmetic circuits incrementally. The proposed approach is based on partitioning the input space of the design by applying certain constraints on primary inputs to solve the equivalence checking problem for each input constraint. If set $\mathbb{M} = \{0,1\}^n$ shows all input combinations of a design with input bits $\{x_0, x_1, ..., x_{n-1}\}$ and if specification ($\mathbb{S}$) and implementation ($\mathbb{I}$) are equivalent for all combinations of ($\mathbb{S} \overset{\mathbb{M}}{\equiv} \mathbb{I}$), they should also be equivalent for any input combinations that belongs to $\mathbb{M}$ ($\forall M \subset \mathbb{M}, \quad \mathbb{S} \overset{M}{\equiv} \mathbb{I}$). If the implementation is buggy, at least one of the intermediate reductions will result in a non-zero remainder.

Clearly, it is not feasible to repeat the equivalence checking procedure for all 2^n input combinations if the design contains n bits of primary inputs. In the existing methods, the inputs are represented by abstract symbols that can get any values. However, we propose an input space partitioning method that is based on keeping some of the primary inputs in symbolic form and assigning Boolean values (either zero or one) to the rest of the primary inputs. This approach expedites the remainder generation time, and it also reduces the number of terms in the remainder and makes it possible to generate a remainder irrespective of the location of the bug. In other words, this approach prevents the remainder's term explosion effect.

Algorithm 8 shows our input partitioning approach. Given the set of primary inputs K with a particular order (order selection will be discussed in Section 5.1.2), the algorithm returns n different constraints on primary inputs where n is the number of primary inputs. Initially, the algorithm sets all of the inputs to zero except the first input in set K which is kept in the symbolic form, and the algorithm adds them to the set of results $\mathbb{M}$ (lines 5-8). In the next step, it keeps the first input in the symbolic form and sets the second input of the ordered set as '1', and sets other inputs to '0', and adds the constraints to the result. This process continues until all of the inputs are kept in their symbolic form except the last one which is set to true. The variable *index* presents the index of primary inputs that should be assigned to true (line 11). The variables before the index variable are kept in their symbolic form, and variables that come after the index are assigned to false (lines 12-15). In each iteration, the *index* variable is updated (line 16). The algorithm returns the set of constraints as output. This algorithm guarantees (see the proof of Theorem 1) that the entire inputs' space is covered since all of the combinations of primary inputs are considered (each input bit is assigned to either one, zero or kept in the symbolic form which can take both values).

Example 1: Assume that we want to partition the input space of the 2-bit multiplier shown in Figure 5-3 using Algorithm 8. Suppose that primary inputs are given in the

Algorithm 8: Algorithm for Generating of Input Constraints

```
1: Input: Primary inputs $K$
2: Output: Set of Constraints Map $\mathbb{M}$
3: new map $M = \{\}$; $n = sizeOf(K)$
4: $M.add(0, K[0])$
5: for $i = 1; i < n; j++$ do
6:    $M.add(i, false)$
7: end for
8: $\mathbb{M}.add(M)$, $index = 1$
9: for $i = 0; i \leq n; i++$ do
10:    $M = \{\}$
11:    $M.put(index, true)$
12:    for $j = 0; j < index; j++$ do
13:       $M.add(j, K[j])$
14:    end for
15:    for $j = index + 1; i \leq n; j++$ do
16:       $M.put(j, false)$
17:    end for
18:    $index++$
19:    $\mathbb{M}.add(M)$
20: end for
21: Return: $\mathbb{M}$
```

following order: $\{A_0, A_1, B_0, B_1\}$. Table 5-1 shows the four different constraints on primary inputs. It can be easily verified that these four constraints cover the entire primary inputs' space. The first and second rows cover two combinations each, the third row covers four combinations, and the last row covers eight combinations. Therefore, it covers all sixteen combinations in Table 8. ■

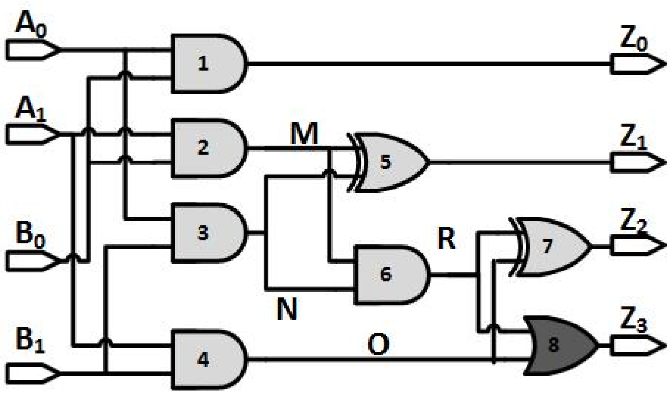


Figure 5-3. Faulty netlist with one bug (gate 8 should have been an AND)

Table 5-1. Different constraints on primary inputs of a 2-bit multiplier.

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

Theorem 1. *A constraint table with n variables (n rows) effectively captures 2^n input sequences.*

Proof. The first row of the constraint table covers two of the input sequences since the first variable is kept as its symbolic form (which it can be either 0 or 1) and other variables are assigned to 0s. Similarly, the second row also covers two combinations as the first variable is in its symbolic form and the second variable is fixed to 1 and the rest of the variables are assigned to zero. Likewise, in the row i where $i \neq 1$, i variables are in their symbolic forms and one variable is assigned to 1 and rest of the variables are assigned to 0. Therefore, row $i \neq 1$ captures 2^{i-1} input sequences. Therefore, for n ($n > 1$) rows we have:

$$2 + \sum_{i=2}^{n} 2^{i-1} = 2 + (2^n - 2) = 2^n$$

We propose an incremental equivalence checking method using the constraints computed based on Algorithm 8. The original equivalence checking problem is mapped to n equivalence checking sub-problems where the specification and implementation polynomials are updated by applying the corresponding constraints. In each sub-problem, a new set of implementation polynomials is computed based on propagating the integer values of the corresponding constraint and considering them while constructing polynomials of each gate and each fanout-free region. Specification polynomial is also updated by applying the conditions of primary inputs in the original specification polynomial. In each sub-problem, the corresponding specification polynomial is reduced over the related implementation polynomials. If the remainder is non-zero, the given constraint manifests some bugs in the design. The implementation and specification of an arithmetic circuit are equivalent if remainders of each of the n sub-problems is computed as a zero remainder.

Algorithm 9: Incremental Equivalence Checking Algorithm

```
1: Input: Input constraint $M_i$, specification polynomial $f_{spec}$, Gate-level netlist $C$
2: Output: Remainder $r$ if the implementation is faulty
3: $f_{spec_i}$ =findSpecificationPolynomial($f_{spec}$, $M_i$)
4: $\mathbb{F}_i$ =findImplementationPolynomials($C$, $M_i$)
5: $r_i$ = reduction of $f_{spec_i}$ over $f_j s \in \mathbb{F}_i$
6: if ($r_i! = 0$) then
7:     Implementation is buggy
8:     return $r_i$
9: end if
10: Return: 0 {correct implementation for constraint $M_i$}
```

Algorithm 14 shows the procedure of incremental equivalence checking for one iteration. It takes input constraints M_i, original specification polynomial f_{spec} as well as partitioned gate-level netlist C as inputs. It evaluates whether specification f_{spec} and implementation C are equivalent considering the constraint M_i. The algorithm returns a counterexample in case of mismatch. The algorithm consists of n iterations each responsible for one inputs' constraints. Specification polynomial f_{spec} is updated based on the input constraints (line 4). Implementation polynomials corresponding to fanout-free region's of C are also reconstructed by applying the constraints. Note that, the polynomial of a fanout-free region is reconstructed if at least condition of one of the region's inputs is different. Otherwise, the polynomial computed in previous iteration is reused. The equivalence checking uses Gröbner basis reduction to reduce f_{spec_i} over implementation polynomials $\mathbb{F}_i$ to find a non-zero remainder if a bug exists in the implementation.

Example 2: Consider the 2-bit multiplier shown in Figure 5-3. We want to apply the incremental equivalence checking approach of Algorithm 14 using all of the input constraints shown in Table 5-1 to verify the correctness of the implementation. Four iterations of the equivalence checking process are needed as shown in Equation 5–1 to manifest the effect of the bug. Specification and implementation polynomials are updated using each constraint. For instance, polynomial of gate 3 is computed as: $N = A_0 + B_1 - A_0.B_1 = A_0$ as B_1 is considered zero in the first iteration (first row of the Table 5-1). Since the last iteration generates a non-zero remainder, the implementation is faulty. ■

$$
\begin{aligned}
&\mathbb{F}_1 = \{Z_0 = 0, M = 0, N = 0, O = 0, R = 0, Z_1 = 0, Z_2 = 0, Z_3 = 0\}\\
&f_{spec_1} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0\\
&step_{1_1}(remainder) : 0\\
&\mathbb{F}_2 = \{Z_0 = A_0, M = 0, N = 0, O = 0, R = 0, Z_1 = 0, Z_2 = 0, Z_3 = 0\}\\
&f_{spec_2} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0\\
&step_{1_2}(remainder) : 0\\
&\mathbb{F}_3 = \{Z_0 = A_0, M = A_1, N = 0, O = 0, R = 0, Z_1 = M, Z_2 = 0, Z_3 = 0\}\\
&f_{spec_3} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 2.A_1 - A_0\\
&step_{1_3} : 2.Z_1 + Z_0 - 2.A_1 - A_0\\
&step_{2_3}(remainder) : 0\\
&\mathbb{F}_4 = \{Z_0 = A_0.B_0, M = A_1.B_0, N = A_0, O = A_1, R = M.N,\\
&Z_1 = A_1, Z_2 = M + N - 2.M.N, Z_3 = R + O - R.O\}\\
&f_{spec_4} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 4.A_1 - 2.A_1.B_0 - 2.A_0 - A_0.B_0\\
&step_{1_4} : 8.R + 8.O - 16.R.O + 2.Z_1 + Z_0 - 4.A_1 - 2.A_1.B_0 - 2.A_0 - A_0.B_0\\
&step_{2_4} : 8.M.N + 8.O - 16.M.N.O + 2.M + 2.N + Z_0 - 4.A_1 - 2.A_1.B_0\\
&- 2.A_0 - A_0.B_0\\
&step_{3_4}(remainder) : 8.A_1 - 8.A_1.A_0.B_0
\end{aligned}
\tag{5–1}
$$

As shown in Equation 5–1, the remainder is generated using the last constraint of Table 5-1 (worst case). However, the complexity of equivalence checking using the given constraint is lower than existing approaches. The generated remainder also has lower complexity compared to with the original remainder that can be achieved with existing methods ($r = 8.A_1.B_1 - 8.A_1.A_0.B_0.B_1$).

The merit of this approach can be observed for verifying complex and buggy implementation since the size of the remainder's terms are reduced by assigning the design variables to either zero or one. Therefore, the possibility of term explosion is drastically reduced. On the other hand, if the implementation is correct, all of the n iterations should be performed. However, the time complexity does not grow n times since the time and memory complexities of most of the iterations are negligible.

5.1.2 Ordering of Primary Inputs

Ordering of primary inputs to produce inputs' constraints impacts the performance of the proposed verification approach of Section 5.1.1 when the implementation is buggy. The size of the remainder depends on the location of the bugs. In this section, we explore a set of ordering of primary inputs to facilitate remainder generation for hard-to-detect bugs and prevent term explosion effect.

The remainder grows through the procedure of reduction of f_{spec} over implementation polynomials. The very first time that the functionality of the buggy gate is involved in the intermediate steps of reduction of f_{spec}, the core of the remainder is formed by terms showing the difference of the faulty functionality from the expected functionality (δ). Terms of the remainder grow gradually during the reduction by substituting terms of the δ with the functionality of gates in the input cone of the faulty cone. Therefore, this approach is extremely helpful while verifying integer arithmetic circuits which contain long carry chain and the functionality of primary outputs is dependent on earlier stages of the design. We propose to start the incremental verification using the constraints which keep the most significant inputs' bits in symbolic forms and assign other bits to zero. This approach manifests the existing bugs with a smaller and more efficient remainder and makes the debugging possible. We propose to interleave bits of different primary inputs and sort inputs' bit in the set K in descending order (most significant bits get higher order) while constructing inputs' constraints using Algorithm 8.

Table 5-2. Input constraints to efficiently verify and debug faulty circuit shown in Figure 5-3. The Most significant bits have priority.

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

Example 3: To efficiently verify the implementation of a 2-bit multiplier shown in Figure 5-3, we use Algorithm 8 with the following order of primary inputs: $\{A_1, B_1, A_0, B_0\}$. Algorithm 8 generates input's constraints as shown in Table 5-2. Equation 5–2 shows the steps of the verification. As it can be observed from Equation 5–2, the complexity of the remainder has reduced significantly in comparison with the remainder of Example 2 ($8.A_1 - 8.A_1.A_0.B_0$) as well as original remainder of existing approaches ($r = 8.A_1.B_1 - 8.A_1.A_0.B_0.B_1$). Different ordering improves the complexity of the incremental verification approach and enables generation of a more efficient remainder for the same bug. ■

$$\begin{aligned}
&\mathbb{F}_1 = \{Z_0 = 0, M = 0, N = 0, O = 0, R = 0, Z_1 = 0, Z_2 = 0, Z_3 = 0\} \\
&f_{spec_1} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 \\
&step_{1_1}(remainder) : 0 \\
&\mathbb{F}_2 = \{Z_0 = 0, M = 0, N = 0, O = A_1, R = 0, Z_1 = 0, Z_2 = A_1, Z_3 = A_1\} \\
&f_{spec_2} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 4.A_1 \\
&step_{1_2} : 2.Z_1 + Z_0 + 8.A_1 \\
&step_{2_2}(remainder) : 8.A_1
\end{aligned} \tag{5–2}$$

5.1.3 Incremental Debugging

The generated remainder and the corresponding constraints can be used to debug the faulty design more effectively based on the approach presented in [52]. Our approach is orthogonal to [52] and can be used on top of it. To generate directed tests to activate the existing fault, assignment to the remainder' variables can be performed such that the integer value of the remainder becomes non-zero. Smaller remainder needs less effort to generate directed tests. The generated test and associated constraints are used to find faulty outputs and localize the source of the bug. The remainder contains terms which show the difference in the functionality of the faulty gate with the expected correct gate based on the functionality of the gate's inputs. Therefore, for each suspicious gate, two patterns are constructed based on Table 5-3 to detect and correct the source of the bug.

Using the incremental debugging, it is possible that some inputs of gates are equal to zero. Therefore, some suspicious AND gates may have two equal patterns during pattern construction. It can be observed from Table 5-3, while generating patterns for a suspicious AND gate, if either input a or b is equal to zero, then P_1 and P_2 will be equal. Only in this case, the equivalence checking should be repeated in order to find the solution of the faulty AND gate (whether the correct gate should be an OR gate or a XOR gate).

Table 5-3. Templates can be caused by gate misplacement error

Faulty Gate	Appeared Remainder's Pattern	Correct Gate
AND (a,b)	P_1 : -a-b+2.a.b	OR (a,b)
	P_2 : -a-b+3.a.b	XOR (a,b)
OR (a,b)	P_1 : a+b-2.a.b	AND (a,b)
	P_2 : a.b	XOR (a,b)
XOR (a,b)	P_1 : a+b-3.a.b	AND (a,b)
	P_2 :-a.b	OR (a,b)

Algorithm 10 shows an overview of our proposed incremental approach. The algorithm includes four key parts: i) finding an efficient order of the primary inputs (line 3); ii) partitioning the inputs' space into different constraints based on the given order (line 4 is Algorithm 1); iii) incremental equivalence checking (line 6 is Algorithm 2); and iv) incremental debugging (lines 9-12) which can be performed using [52].

Example 4: Considering the faulty implementation shown in Figure 5-3 and remainder $r = 4.A_1$, the only assignment that makes r non-zero is $A_1 = 1$. Considering the other constraints that generates the remainder, $A_1, B_1, A_0, B_0 =$ "$11XX$" is a directed test to activate the fault. The test activates the effect of the bug in primary output Z_3. Therefore, gates $2, 3, 4, 6, 8$ are suspicious. Patterns are constructed for each of the gate as $2(P_1 = P_2 = 2.A_1), 3(P_1 = P_2 = 1), 4(P_1 = P_2 = 4.A_1), 6(P_1 = P_2 = 0), 8(P_1 = 8.A_1, P_2 = 0)$. Therefore, gate 8 is faulty and it should be replaced with an AND gate. Note that the weight of each gates' output is computed by considering known weight of primary inputs and outputs, and traversing in both backward and forward directions while propagating the weights based on the approach outlined in [58]. ■

Algorithm 10: Incremental Debugging Algorithm

```
1: Input: Specification polynomial $f_{spec}$, Gate-level netlist $C$, Primary inputs $PI$
2: Output: Potential faulty gate and its solution
3: $K$ = OrderPrimaryInputs(PI)
4: $\mathbb{M}$=GenerateInputConstraints(K)
5: for each input constraints $M_i \in \mathbb{M}$ do
6:    $r_i$=IncrementalEquivalenceChecking($M_i, f_{spec}, C$)
7:    if ($r_i! = 0$) then
8:       $\mathbb{T}$=GenerateDirectedTests($r_i$)
9:       $\mathbb{G}$=BugLocalization($\mathbb{T}, C$)
10:      $(G, s)$= BugDetectionAndCorrection($\mathbb{G}$)
11:      gate $G$ is buggy and $s$ is the solution
12:   end if
13: end for
14: Return: Implementation is correct {if none of the constraints finds a non-zero
    remainder}
```

Our proposed approach can also be beneficial to debug multiple errors since our incremental debugging method has the ability to differentiate the effect of each bug in one specific remainder. For example, suppose that two bugs, shown in Figures 4-4 and 5-3, are inserted in the same circuit. Therefore, the process shown in Equation 5–2 can show the effect of the last stage bug. Using the procedure shown in Example 4, the bug can be fixed. After fixing the first bug, if we continue the incremental debugging procedure, the effect of the second bug will be manifested in remainder $r2 = +1 - 2.A_0$. Using the same method shown in Example 1, the second bug also can be detected and fixed. Therefore, both bugs in the design can be fixed.

5.2 Experiments

5.2.1 Experimental Setup

Incremental equivalence checking and debugging algorithms were implemented using a Java program and experiments were performed on an Intel Xeon Processor with 16 GB memory. We have tested our approach on post-synthesized gate-level integer arithmetic circuits that implement adders and multipliers. The post-synthesized integer arithmetic circuits are more difficult to verify and debug due to their optimized architectures and their carry chains. The designs were synthesized using Xilinx synthesis tool. We consider gate misplacement that changes the functionality of the design as our fault model. To illustrate that our debugging approach is not dependent on the location of the bug, we partitioned the implementation in four levels and several gates from these levels were randomly replaced with a faulty gate to create erroneous implementations. Inputs' constraints are generated automatically using a program. The order of constraints is determined based on the approach presented in Section 5.1.2. In order to generate the remainder, we have created three different threads: i) using no input constraints (which is similar to [34]), ii) using input constraints starting from the least significant input bits (as described in Section 5.1.1), and iii) using input constraints starting from the most significant bits (as explained in Section 5.1.2). We have considered the fastest time for remainder generation among these three threads for the reported time in equivalence checking. In other words, when any one of these threads generates a remainder, the other two threads are terminated. We compared our equivalence checking and debugging results with state-of-the-art approaches [34, 52].

5.2.2 Results

Table 5-4 presents the results of our incremental equivalence checking and debugging approaches. The first column presents the type of the benchmarks which are either two-input ripple carry adders or two-input array multipliers. The second and third columns indicate the input size ($firstOperand \times secondOperand$) and design size (number of gates), respectively. The fourth column shows the location of the bug. We partition the implementation into four levels, e.g. "$0 - 1/4$" refers to the gates closest to the primary inputs and "$3/4 - 4/4$" refers to the gates which are placed in the deeper stages of the design (closest to the primary outputs). The fifth column shows the equivalence checking result using Z3 SMT solver [43]. To use a SMT solver, we have converted polynomials (specification and implementation polynomials) into SMT solver format. The sixth column presents the equivalence checking (run-time in seconds) results of [34]

Table 5-4. The results of the proposed incremental equivalence and debugging approaches for integer arithmetic circuits. TO = timeout after 5 hours; MO = memory out of 16 GB.

Type	Size	#Gates	Bug. Loc.	Equivalence checking (s)				Debugging (s)								
								[52]				Our Approach				Imp.
				Z3	[34]	Ours	Imp.	TG	BL	DC	Total	TG	BL	DC	Total	
Post-Syn Mult.	8x8	368	0 – 1/4	23.35	$\underline{0.05}$	0.05	1x	0.08	0	0.05	0.13	0.01	0	0.02	0.03	4.3x
			1/4 – 2/4	$\underline{22.83}$	45.31	0.05	456.6x	5.17	0.01	3.64	8.82	0.01	0	0.02	0.03	294x
			2/4 – 3/4	$\underline{22.59}$	TO	0.04	564.7x	_	_	_	_	0.01	0	0.06	0.07	∗
			3/4 – 4/4	$\underline{23.13}$	TO	0.03	771x	_	_	_	_	0.01	0	0.07	0.08	∗
	16x16	1.6K	0 – 1/4	107.1	$\underline{0.51}$	0.51	1x	0.34	0.01	0.69	1.04	0.02	0	0.03	0.05	20.8x
			1/4 – 2/4	$\underline{104.5}$	TO	0.12	870x	_	_	_	_	0.03	0.01	0.2	0.24	∗
			2/4 – 3/4	$\underline{105.4}$	MO	0.14	752.8x	_	_	_	_	0.01	0.01	0.3	0.32	∗
			3/4 – 4/4	$\underline{109.6}$	MO	0.15	730.6x	_	_	_	_	0.02	0.02	0.45	0.49	∗
	32x32	7K	0 – 1/4	MO	$\underline{1.57}$	1.57	1x	0.75	0.1	5.27	6.12	0.1	0.02	0.1	0.22	27.8x
			1/4 – 2/4	MO	MO	1.09	∗	_	_	_	_	0.1	0.03	0.31	0.44	∗
			2/4 – 3/4	MO	MO	0.5	∗	_	_	_	_	0.1	0.04	4.0	4.14	∗
			3/4 – 4/4	MO	MO	0.2	∗	_	_	_	_	0.03	0.1	5.64	5.77	∗
	64x64	28K	0 – 1/4	MO	$\underline{16.43}$	16.43	1x	2.7	5	19.21	26.91	0.1	0.1	0.9	1.1	24.5x
			1/4 – 2/4	MO	MO	24.16	∗	_	_	_	_	0.3	0.1	2.6	3.0	∗
			2/4 – 3/4	MO	MO	1.35	∗	_	_	_	_	0.3	0.2	14.7	15.2	∗
			3/4 – 4/4	MO	MO	0.72	∗	_	_	_	_	0.1	1.6	8.2	9.9	∗
	128x128	132K	0 – 1/4	MO	$\underline{62.23}$	62.23	1x	4.7	28	278.4	311.1	1.1	0.3	3.5	4.9	63.5x
			1/4 – 2/4	MO	MO	256.24	∗	_	_	_	_	1.3	0.7	21.0	23.0	∗
			2/4 – 3/4	MO	MO	37.67	∗	_	_	_	_	0.9	2.2	50.8	53.9	∗
			3/4 – 4/4	MO	MO	20.41	∗	_	_	_	_	0.4	3.9	38.1	42.4	∗
Avg.				> 64.8	> 21.01	21.2	> 377x	> 2.2	> 5.5	> 51.2	> 59x	0.25	0.45	7.6	8.3	> 72.5
Post-Syn Adder	64x64	573	0 – 1/4	51.36	$\underline{0.02}$	0.02	1x	0.69	0	0.31	0.82	0.01	0	0.01	0.02	68.3x
			1/4 – 2/4	$\underline{38.39}$	TO	1.04	36.91	_	_	_	_	0.01	0	0.02	0.03	∗
			2/4 – 3/4	$\underline{32.61}$	TO	0.71	45.9x	_	_	_	_	0.01	0	0.07	0.08	∗
			3/4 – 4/4	$\underline{30.51}$	TO	0.12	254.2x	_	_	_	_	0.01	0	0.07	0.08	∗
	128x128	1.2K	0 – 1/4	101.1	$\underline{0.17}$	0.17	1x	1.99	0.01	0.5	2.5	0.01	0	0.01	0.02	138.8x
			1/4 – 2/4	$\underline{115.8}$	TO	4.2	27.6x	_	_	_	_	0.01	0	0.05	0.06	∗
			2/4 – 3/4	$\underline{116.9}$	MO	2.8	41.7x	_	_	_	_	0.01	0	0.06	0.07	∗
			3/4 – 4/4	$\underline{115.9}$	MO	1.04	111.4x	_	_	_	_	0.01	0.01	0.07	0.09	∗
	256x256	2.3K	0 – 1/4	158	$\underline{3.84}$	3.84	1x	3.35	0.1	1.51	4.96	0.01	0.01	0.03	0.05	107.8x
			1/4 – 2/4	$\underline{252.5}$	TO	21.6	11.7x	_	_	_	_	0.01	0.01	0.2	0.22	∗
			2/4 – 3/4	$\underline{281.5}$	MO	31.3	8.9x	_	_	_	_	0.01	0.01	0.14	0.16	∗
			3/4 – 4/4	$\underline{301.7}$	MO	4.4	68.5x	_	_	_	_	0.01	0.01	0.3	0.32	∗
Avg.				133.0	> 1.3	6	48.3x	> 2	> 0.1	> 0.7	> 2.7	0.01	0.01	0.1	0.1	> 105x

"*" indicates our approach works but existing method fails. "_" shows the cases when test generation, bug localization/correction cannot be done due to lack of the remainder.

for a faulty design. The seventh column indicates the time of our proposed incremental equivalence checking method which includes the time of the inputs' space partitioning as well as Algorithm 14. The eight column presents the improvement (Imp.) provided by our approach in comparison with the best result (indicated using underscore) of existing approach shown in fifth and sixth columns. In the table, "∗" indicates that our approach performs well, whereas the existing approach [34] fails. The SMT solver tries to find a counterexample when implementation and specification are not equal. As it is shown, the SMT solver cannot find a counterexample for large designs. Moreover, there is no efficient and fully automatic debugging approach for fixing the bug using SMT solvers. It can be observed that when a bug exists on deeper stages of the design, this approach fails even for very small benchmarks. When the bug is close to the primary inputs, the incremental approach can take more time to finish since it goes through several iterations if they result in a zero remainders. However, when the bug is not near primary inputs, our approach not only makes the checking possible but also is faster by several orders-of-magnitude.

The next four columns show the time (in seconds) required for test generation (TG), bug localization (BL), debug (DC) and total (TG+BL+DC), respectively, using [52]. The subsequent four columns shows the same features using our proposed approach. The test generation time is dependent on the number of terms in the reminder. Since our approach generates more compact remainder, test generation time has improved by several orders of magnitude. If a non-zero remainder can be obtained using a smaller number of inputs' constraints, the remainder will be more compact. Since we use the constraints' order where most significant bits comes first, if the bug exists close to the primary outputs, the chance of obtaining a more compact remainder is more and test generation time is reduced. The results show that our approach requires less time to perform bug localization since the size of the generated remainder is small, and as a result, the number of directed tests are less. Moreover, the results show the effectiveness of incremental debugging approach based on required time for bug detection and correction compared to [52]. If a bug is located close to primary outputs, the number of suspicious gates is increased, therefore, the time for bug correction and detection increases. In the table, "_" indicates that test generation, bug localization, and bug correction are not possible due to lack of the remainder. Finally, the last column presents the improvement (Imp.) provided by our incremental debugging approach. Clearly, our proposed approach can drastically ($>$ 70 times one average) reduce the overall debugging effort. Most importantly, it is able to debug hard-to-detect errors when existing state-of-the-art methods fail.

Table 5-5 presents the equivalence checking time for correct implementations of different designs. We have compared our proposed framework with Z3 SMT solver and [34]. As it can be observed, our method outperforms the Z3 and its performance is comparable with [34]. Table IV and V demonstrate that our approach performs well irrespective of whether the implementation is buggy or not.

Table 5-5. The equivalence checking time for correct designs.

Benchmark	Size	Z3 SMT Solver	[5]	Our Approach	Imp. on Z3
Post-Syn. Multiplier	8x8	47.81	0.04	0.05	957.04x
	16x16	TO	0.11	0.16	*
	32x32	MO	0.42	0.43	*
	64x64	MO	2.50	2.68	*
	128x128	MO	19.25	19.77	*
Post-Syn. Adder	64x64	31.84	0.08	0.1	318.4x
	128x128	190.61	0.4	0.44	433.27x
	256x256	513.98	0.84	0.88	584.07

Our experimental results highlight three important aspects of our debugging approach. First, using the inputs' constraints as well as the incremental debugging address the scalability issues of the existing arithmetic circuits' equivalence checking methods. Second, our incremental equivalence checking method enables more compact remainder generation that can improve the required time for test generation, bug localization and bug detection. Finally, the debugging approach automatically and efficiently detects and corrects unknown bugs regardless of its complexity and location in the design.

5.3 Summary

In this chapter, we presented an incremental equivalence checking and debugging framework for arithmetic circuits. The proposed approach made three important contributions. It partitions the primary inputs' space of the design in order to solve the verification and debug problems in the increasing order of the design complexity. Moreover, it developed an incremental equivalence checking algorithm to enable generation of compact remainders. Finally, the proposed input ordering and incremental debugging enabled efficient bug detection and correction. Our experimental results demonstrated that our incremental verification framework is several orders-of-magnitude faster than existing state-of-the-art approaches.

CHAPTER 6
TROJAN LOCALIZATION USING SYMBOLIC ALGEBRA

Growing reliance on reusable hardware Intellectual Property (IP) blocks, severely affects the security and trustworthiness of System-on-Chips (SoCs) since untrusted third-party vendors may deliberately insert malicious components to incorporate undesired functionality. Malicious implants may also work as hidden backdoor and leak protected information. In this chapter, we propose an automated approach to identify untrustworthy IPs and localize malicious functional modifications (if any). The technique is based on extracting polynomials from gate-level implementation of the untrustworthy IP and comparing them with specification polynomials. The proposed approach is applicable when the specification is available. Our approach is also useful when a golden design has gone through non-functional transformations such as synthesis, and we would like to ensure that the modified design is trustworthy. Our approach is scalable due to manipulation of polynomials instead of BDD-based analysis used in traditional equivalence checking techniques. Experimental results using Trust-HUB benchmarks demonstrate that our approach improves both localization and test generation efficiency by several orders of magnitude compared to the state-of-the-art Trojan detection techniques.

Figure 6-1 presents the overview of our proposed methodology. We extract a set of polynomials from the specification ($\mathbb{S}$). We also derive a set of polynomials ($\mathbb{I}$) from the implementation. Finally, we check the equivalency between two sets $\mathbb{S}$ and $\mathbb{I}$ based on Gröbner Basis Reduction. Each of the polynomials from the specification f_{spec_i} is reduced over a set of corresponding polynomials $\mathbb{I}$ and a set of remainders $\mathbb{R}$ is generated. From symbolic computer algebra, it is known that when $r_i = 0$, gates in Rg ((set of gates that contribute in reduction of polynomial f_{spec_i} is called region Rg)) have successfully implemented f_{spec_i} and it guarantees that all gates in Rg are safe [62]. Any $(r_i \neq 0) \in \mathbb{R}$ shows a suspicious functionality in the corresponding region Rg and all of the gates in Rg are suspicious candidates. The malicious nodes can be pruned by removing the safe gates

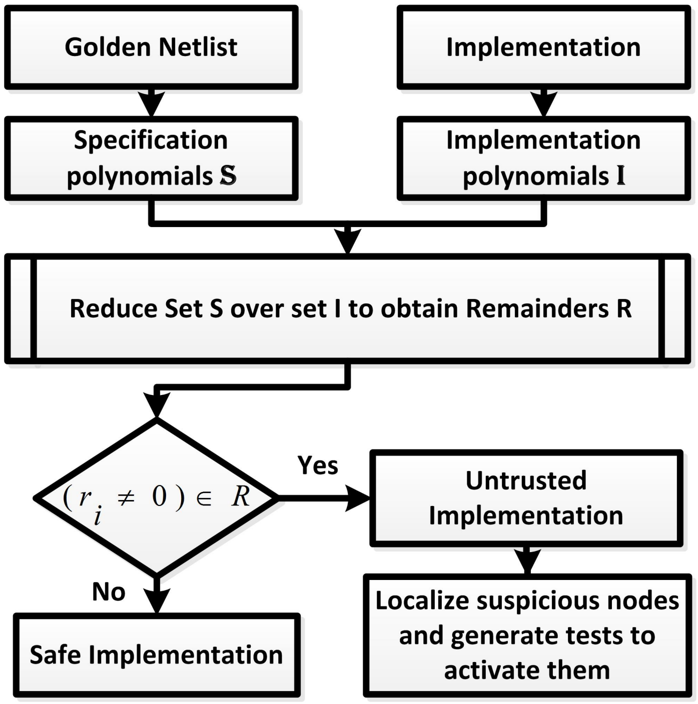

Golden Netlist
Implementation
Specification polynomials S
Implementation polynomials I
Reduce Set S over set I to obtain Remainders R
$(r_i \neq 0) \in R$
Yes
Untrusted Implementation
No
Safe Implementation
Localize suspicious nodes and generate tests to activate them

Figure 6-1. The proposed hardware Trojan localization flow

from the suspicious candidates. When all of r_i's are equal to zero, the implementation is Trojan free. The proposed method can recognize the Trojan-free implementation from the Trojan-inserted one. Our method reports a few gates to indicate the presence of a malicious activity (change of functionality) in the implementation. Since the number of malicious gates is very small, our approach is amenable for an exhaustive test generation to activate the Trojan. Our method is applied on Trust-HUB benchmarks [69] and the experimental results show the effectiveness of our approach compared to existing methods.

The remainder of this chapter is organized as follows. Section 6.1 discusses our framework for hardware Trojan localization and detection. Section 6.2 presents our experimental results. Finally, Section 6.3 concludes this chapter.

6.1 Trojan Detection and Localization

In order to trust an IP block, we have to make sure that the IP is performing exactly the expected functionality. The approach presented in Section 3 can be extended to find whether a hardware Trojan, which changes the functionality, has been inserted in a combinational arithmetic circuit. However, applying the same approach on general IPs is limited due to several reasons. First, it is possible that the specification of a general circuit cannot be described as one simple polynomial. Second, the circuit may not be acyclic and loops may exist due to their sequential nature. Third, unrolling may increase the complexity of the problem so the reduction of f_{spec} over implementation polynomials will face polynomial terms explosion. Finally, the Trojan activation may require extremely large number of unrolling steps which may be practically infeasible and also there is no specific information on after how many cycles Trojan will be activated. In order to address these challenges, we present a method to generate polynomials in an efficient way and use them in our proposed algorithm to localize and detect Trojans in third-party IPs. To the best of our knowledge, our proposed approach is the first attempt in utilizing scalable equivalence checking using polynomial manipulation for localization of hardware Trojans. The reminder of this section describes the three important tasks in our framework: polynomial generation, Trojan localization, and test generation for Trojan detection.

6.1.1 Polynomial Generation

Suppose that we have two versions of a design, one is a verified IP (specification) and the other is an untrusted third-party IP (implementation) after performing non-functional transformations. Our goal is to detect whether an adversary has inserted hard-to-detect hardware Trojan during non-functional changes and has made undesired functional changes. For example, a design house may send their RTL design for synthesis or adding low-power features to a third party vendor. Once the third-party IP comes back (after synthesis or other functionality-preserving transformations), it is crucial to ensure the trustworthiness of these IPs.

In the method presented in Section 3, specification is modeled as one polynomial; however, here we generate a set of polynomials $\mathbb{S}$ representing the functionality of the golden IP to be able to apply Gröbner basis theory for hardware Trojan localization problem. The specification is partitioned into several regions and each region is converted to a polynomial. The output of each region is either inputs of a flip-flop (clock, enable, reset and etc.) or one of the primary outputs. The inputs of a region are either from primary inputs or inputs/outputs of flip-flops. In other words, we generate polynomials for regions which are limited to flip-flops' boundaries. Then, corresponding equations (based on Equation 3-1 in Section3.2) of gates inside a region are combined together to construct one polynomial representing the functionality of the region.

Algorithm 11: Polynomial generation algorithm

```
1: Input: Circuit Graph $Gr$, $L_{out}$ and $L_{in}$
2: Output: Polynomials $\mathbb{S}$
3: Region = {}
4: for each gate $g_i \in Gr$ where its output $\in L_{out}$ do
5:    Region.add($g_i$)
6:    for all inputs $g_j$ of $g_i$ do
7:       if !($g_j \in L_{in}$) then
8:          Region.add($g_j$)
9:          Call recursively for inputs of $g_j$ over $Gr$
10:      end if
11:   end for
12:   $f_i$ = convertToPolynomial(Region)
13:   $\mathbb{S} = \mathbb{S} \cup f_i$
14:   Region = {}
15: end for
16: Return: $\mathbb{S}$
```

Algorithm 11 shows how we extract set $\mathbb{S}$. The specification is converted to a graph where each vertex is a gate (g_i). The algorithm takes the circuit graph Gr, list (L_{out}) of allowed output variables (flip-flops' inputs and primary outputs) and list (L_{in}) of allowed input variables of a region as inputs and returns a set of polynomials $\mathbb{S}$ as its output. The algorithm chooses a gate for which output belongs to L_{out} and goes backward recursively until it reaches the gate g_j, whose input comes from one of the variables from L_{in} (line 5-10). The algorithm marks all the visited gates as a "*Region*". The selected region may contain all of the basic gates except flip-flops. Then, the *Region* is converted to a polynomial f_i by combining corresponding polynomials of the gates residing in *Region*, f_i is added to set $\mathbb{S}$ (line 11-12).

Example 1: Suppose that the circuit shown in Figure 6-2 is a part of a verified IP block and we want to use it as our specification. Algorithm 11 is applied on it and the polynomials are shown as: $\mathbb{S} = \{f_{spec_1} : n_1 - (-2.A.n_2 + n_2 + A), f_{spec_2} : Z - (1 - n_1.B)\}$. Since the circuit shown in Figure 6-2 contains one primary output and one flip-flop, the Algorithm 11 extracts two specification polynomials for this circuit.

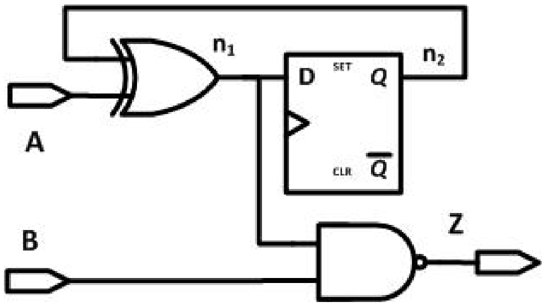


Figure 6-2. A part of a sequential circuit

Similarly, the implementation polynomials $\mathbb{I}$ are driven by modeling every gate except flip-flops from the untrusted design as a polynomial based on Equation 3-1 from Section 3.2 and Algorithm 11. In order to reduce the number of generated implementation polynomials, we partition implementation to fanout-free cones (set of gates that are directly connected together) and convert each fanout-free region as one polynomial. In other words, $\mathbb{I}$ contains a set of polynomials where each polynomial represents a fanout-free cone.

Example 2: The circuit shown in Figure 6-3 is the Trojan-inserted implantation of the specification shown in Figure 6-2 (gate 6 is the Trojan trigger and gate 7 is the payload). Gates in same pattern belong to a common fanout-free cone. As a result, set $\mathbb{I}$ is computed by Algorithm 11. Each polynomials is corresponding to one fanout-free cone.

$$\begin{aligned}\mathbb{I} = \{&n_1 - (n_2.w_4.A - n_2.w_4 + w_4 - n_2.A + n_2),\\ &w_4 - (A - n_2.A),\\ &Z - (n_1.w_4.C.B - n_1.w_4.C - n_1.B + 1)\}\end{aligned} \tag{6–1}$$

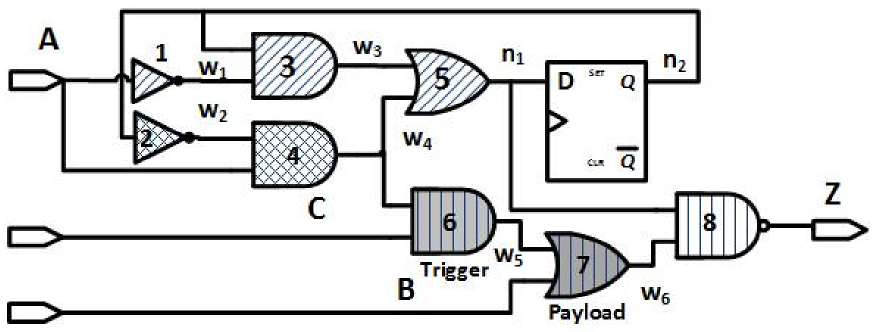


Figure 6-3. A Trojan-inserted implementation of circuit in Figure 6-2

6.1.2 Trojan Localization

We generate the set $\mathbb{S}$ and $\mathbb{I}$ as described in Section 6.1.1. We assume that the name of flip-flops, primary inputs and primary outputs are the same between implementation and specification or the name mapping can be done. We also assume that no re-timing has been performed. These are valid assumptions in many scenarios involving third-party IPs. The equivalence of two sets $\mathbb{S}$ and $\mathbb{I}$ is checked to find any suspicious functionality which may serve as a Trojan.

To detect a Trojan, we need to reduce each polynomial f_{spec_i} from set $\mathbb{S}$ over a subset of polynomials from set $\mathbb{I}$ to check membership of every polynomial f_{spec_i} in Ideal I constructed from polynomials from set $\mathbb{I}$ ($I = < \mathbb{I} >$). To perform that, all of the polynomials from $\mathbb{I}$ are hashed based on their leading terms (which contains a single variable and this variable represents the output of the corresponding gate). Every variable from $f_{spec_i} \in \mathbb{S}$ is replaced with the corresponding functionality of that variable from $\mathbb{I}$ polynomials. The process continues until f_{spec_i} is reduced either to zero polynomial or a remainder polynomial which contains primary inputs as well as flip-flop's inputs/outputs. The non-zero remainder indicates that implementation does not correctly implement the functionality of f_{spec_i} and that part of the implementation is suspicious. Note that, based on Gröbner basis theory, when the remainder is zero for a specific region, we can be certain that the region is safe. In other words, it is not possible for a smart attacker to insert malicious gates in a way that the remainder becomes zero.

Example 3: Consider we want to measure the trust in the circuit shown in Figure 6-3, which is the untrustworthy implementation of design shown in Figure 6-2. Specification Polynomials shown in Example 1 are reduced over implementation polynomials as shown in Equation 6–1. The result of the reduction is stored in set $\mathbb{R}$. Each f_{spec_i} produces one remainder r_i that can be either zero or a non-zero polynomial.

Gates $\{1, 2, 3, 4, 5\}$ implement functionality of an XOR gate (these gates are equivalent to XOR gate shown in Figure 6-2). Thus, the remainder r_1 is zero and it means that the region containing gates $\{1, 2, 3, 4, 5\}$ implements the f_{spec_1} correctly. However, the non-zero remainder r_2 presents the fact that there are malicious components in implementation of f_{spec_2} and the region containing gates $\{2, 4, 6, 7, 8\}$ is suspicious.

$$\begin{aligned}
&f_{spec_1} : n_1 + 2.A.n_2 - n_2 - A \\
&step_{11} : n_2.w_4.A - n_2.w_4 + w_4 + n_2.A - A \\
&step_{12}(r_1) : 0 \\
&f_{spec_2} : Z + n_1.B - 1 \\
&step_{21} : n_1.w_4.C.B - n_1.w_4.C \\
&step_{22}(r_2) : -1.n_1.A.C + n_1.n_2.A.C + A.B.C.n_1 - A.B.C.n_1.n_2
\end{aligned} \tag{6–2}$$

By using the proposed approach, a set of malicious regions are identified. Suppose the adversary inserts some extra flip-flops as part of Trojans. These buggy flip-flops does not have any correspondence in the specification. In other words, there is no f_{spec_i} which describes their inputs' functionality. Therefore, the corresponding region in the implementation is also considered as a suspicious region. However, scan-chain flip-flops can easily be detected and removed from suspicious candidates because of their structures.

The proposed method formally identifies the regions (between flip-flops boundaries) of the implementation that are safe and the regions that have suspicious functionality. The adversary usually insert the Trojan in deep levels of the circuit. Therefore, the regions that actually contain the Trojan can be very large and may include many gates (order of hundreds or thousands of gates). In order to improve our approach further, we propose an algorithm to identify the gates that most likely are responsible for the malicious activity.

Since we know which regions are Trojan-free (based on remainder as zero), we remove the gates which are contributing in construction of these regions from suspicious regions. In other words, we have formally proved that some of the regions are trustworthy so the gates that construct these regions are essential for the correct functionality. The safe gates may be inputs of Trigger or payload gates. However, they do not belong to the set of malicious gates. Using this approach, we are able to prune the suspicious regions to contain very small number of gates. This approach guarantees that all of the Trojan trigger and payload's gates are inside the suspicious region. Algorithm 12 shows the proposed procedure.

Algorithm 12: Hardware Trojan localization algorithm

```
1: Input: Circuit implementation $G_r$, $\mathbb{I}$ and $\mathbb{S}$
2: Output: Suspicious gates $G_t$
3: for each $f_{spec_i} \in \mathbb{S}$ do
4:     $r_i$ = reduction of $f_{spec_i}$ over $f_j s \in \mathbb{I}$
5:     $R_i = R_i \cup all\ g_j s\ where\ f_j = func(g_j)$
6:     mark all $g_i$s as used
7:     if $(r_i! = 0)$ then
8:         $R_{TrjIn} = R_{TrjIn} \cup R_i$
9:     else
10:        $R_{TrjFree} = R_{TrjFree} \cup R_i$
11:    end if
12: end for
13: for each gate $g \in R_{Trjfree}$ do
14:     remove g from $R_{TrjIn}$
15: end for
16: Return: $G_t$ = remaining in $R_{TrjIn} \cup unused\ gates$
```

The algorithm takes the gate-level implementation graph G_r as well as specification and implementation polynomials as inputs, and in case the implementation contains malicious components, it returns a set of suspicious gates as output. The algorithm takes each of specification polynomials and reduces them one by one over corresponding polynomials from set $\mathbb{I}$. Each f_{spec_i} may be reduced using several gates g_j and the result of the reduction is stored in r_i (line 4-5). The used gates are marked to keep track of the gates that are utilized to implement the circuit (line 6). If r_i is equal to zero, it means that all of the g_is are safe and they are stored as safe gates ($R_{TrjFree}$), otherwise, all g_is are stored as suspicious candidates (line 7-11). Every $r_i = 0$ shows that all of the gates used in construction of functionality of the corresponding f_{spec_i} are safe. Therefore, to narrow down the potential suspicious gates, the gates of G_r which appeared in $R_{TrjFree}$ are removed from R_{TrjIn} (line 12-13). Note that, gates in both of $R_{TrjFree}$ and R_{TrjIn} belong to the implementation G_r. All of unused gates should also be considered as malicious candidates, so the union of the remaining gates in the R_{TrjIn} and unused gates are returned as likely malicious gates (G_t). If all of the r_is are zero, the implementation is safe and there is no Trojan inside the implementation.

Algorithm 12 identifies the trust level of a third-party IP and in case of existence of hardware Trojan, it returns a very small number of gates as suspicious candidates. This algorithm guarantees that all of the actual Trojan trigger and payload gates are inside the set G_t.

Example 4: Applying Algorithm 12 on the circuit shown in Figure 6-3 will result in non-zero remainder for region containing gates $\{2, 4, 6, 7, 8\}$. However, the zero remainder of f_{spec_1} shows that gates $\{1, 2, 3, 4, 5\}$ are safe and they are vital to construct the functionality of signal n_1. Therefore, we remove gates $\{2, 4\}$ from potential candidates and gates $\{6, 7, 8\}$ remain as suspicious.

6.1.3 Trojan Activation

As shown in Example 4, the small suspicious region still contains some safe gates which are dedicated to the correct functionality in the absence of the Trojan (in Example 4, gate 8 is benign but it is reported as suspicious node). In other words, these safe gates are only used to construct the functionality of one specific primary output or Flip-Flop's input. Thus, they won't be removed in the process of pruning safe gates from suspicious regions since they are not contributing in functionality of other primary outputs or flip-flop's inputs. To be able to detect the exact gates which are responsible for trigger and payload parts of Trojan, we generate tests to activate the Trojan. Since the number of suspicious gates are small enough, we try to activate each node in the suspicious gates and check whether the generated test activates the Trojan. We use an ATPG to generate the directed tests. If none of the tests detects the Trojan, we generate test to activate two of the nodes at the same time. We continue the process until one of the tests activates the Trojan. The proposed method is shown in Algorithm 13. This approach is feasible due to the fact that the number of suspicious nodes that are reported using our proposed approach is very small.

Algorithm 13: Test generation algorithm

```
1:  **Input:** Suspicious gates $G_t$, Implementation C, Specification S
2:  **Output:** Test vectors $T$
3:  T={}
4:  for each possible trigger scenario $n$ over $G_t$ do
5:      generate test $t_i$ to activate $n$ of nodes
6:      for each possible payload scenario do
7:          propagate effect of $t_i$ to the observable points
8:          if trigger scenario is satisfied then
9:              $T = T \cup t_i$
10:         end if
11:     end for
12: end for
13: Return: $T$
```

Example 5: We are trying to activate the Trojan shown in Figure 6-3. From Example 4, we know that gates $\{6, 7, 8\}$ are suspicious. As shown in Figure 6-3, Trojan will be triggered when output of gate 6 (w_5) becomes true and B is zero at the same time. In other words, gate 8 of the implementation receives one as its second input (w_6) while in the specification, the second input of the NAND gate receives zero. These conditions cause difference between specification and implementation. To propagate the effect of Trojan's condition activation, n_1 should be one since $n_1 = 0$ makes output $Z = 1$ independent of second input's value and it will mask the Trojan effect. The test vectors that activate Trojan are as follows (we assume the initial value of n_2 is equal to 0): $A = 1, B = 0, C = 1$.

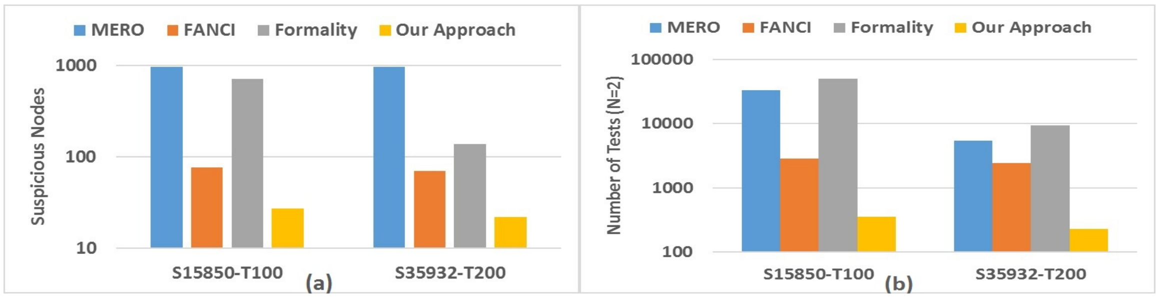


Figure 6-4. (a) Number of suspicious nodes, (b) Number of tests needed to activate Trojans

6.2 Experiments

6.2.1 Experimental Setup

The Trojan localization algorithm was implemented in a Java program and experiments were conducted on PC with Intel Processor E5-1620 v3 and 16 GB memory. We have tested our approach using widely used trust-HUB benchmarks [69] consisting of combinational and sequential Trojan triggers and payloads that change the functionality of the design. The Trojan-Free designs are considered as specification. To show that our methodology is orthogonal to design structures and library format, we synthesized Trojan-inserted benchmarks with Xilinx synthesis tool and used them as implementation (we just map flip-flops' inputs/output names). Specification is partitioned into several regions and each region is represented using one polynomial. These polynomials can be reduced over implementation polynomials independently. Therefore, we used a parallel version of Algorithm 12 to implement our method. We also used logic reduction based rewriting schemes presented in [121] to improve the equivalence checking time. We compared our results with most relevant Trojan localization work [134]. Since our approach essentially performs equivalence checking, we also compared with an equivalence checking tool *"Formality"* [71] which has been designed to check the equivalence between two versions of a design to demonstrate the efficiency of our work. Formality is an commercial tool that tries to detect potential functional changes between two versions of a design when the designers making non-functional changes.

Table 6-1. Trojan Localization using Trust-HUB benchmarks.

Benchmark			FANCI [134]	Formality [71]	Our Approach				False Positive%			Improvement	
Type	#Gates	#TrojanGates	#SuspGates	#SuspGates	#SuspGates	#Spolys	#Ipolys	CPU time(s)	our	[134]	[71]	[134]	[71]
RS232-T1000	311	13	37	214	13	62	186	0.67	0	24	201	*	*
RS232-T1100	310	12	36	213	14	61	189	0.86	2	24	201	12x	100.5x
S15850-T100	2456	27	76	710	27	592	1888	1.13	0	49	683	*	*
S38417-T200	5823	15	73	2653	26	1667	5004	3.12	11	58	2638	5.27x	239.8x
S35932-T200	5445	16	70	138	22	1778	4441	3.18	6	54	122	9x	20.33x
S38584-T200	7580	9	85	47	11	840	3905	4.74	2	76	38	38x	19x
Vga-lcd-T100	70162	5	706	**	22	2426	7572	38.97	17	701	**	41.23x	**

"*" indicates our approach does not produce any false positive gates (infinite improvement).
"**" shows the cases that *Formality* could not detect the Trojans.

Formality compares the points between two designs and tries to match them using different algorithms including name-based matching and non-name based matching algorithms. Based on formality's user guide [72], it first compares the points based on their exact names. Then, it tries to perform case-insensitive name mapping or filtering out some characters. Name matching can also be done through mapping driven/driving nets (name of nets) of points. In the second phase, it attempts to match the remaining unmatched points using topological analysis of the unmatched cones. In other words, it matches two points with different names if they have equivalent structures. The final step is signature analysis which is based on generating functional and topological signatures. Functional signatures use random patterns simulation to generate primary outputs' data or register's output data to match different points. However, if an adversary inserts a hard-to-detect hardware Trojan, signature analysis may incorrectly match points since their simulation result are same. As a result, *Formality* may not be able to detect inserted Trojans (as indicated in Table 6-1). Our proposed method is based on polynomial manipulation of different regions of the circuit and it is not dependent on the simulation or pattern generation. Thus, our method outperforms *Formality* when there are hard-to-activate Trojan in the implementation.

6.2.2 Trojan Localization

Table 6-1 presents results for hardware Trojan localization. The first three columns show the type of benchmarks, number of gates in the circuit, and number of malicious gates (consisting of Trojan trigger and payload), respectively. The fourth column shows the number of suspicious gates reported by *"FANCI"* [134] approach. *FANCI* reports 1% to 8% of circuit nodes as false positive nodes on average (we have reported suspicious nodes as false positive nodes plus actual Trojan gates). The fifth columns shows the number of suspected gates that can be found using *Formality*. It reports some faulty flip-flops or primary outputs which may have different values because of change in the functionality. However, there are so many gates in the cone corresponding to the faulty primary outputs or flip-flops and all of these gates are suspicious. In case Vga-lcd-T100, the Trojan effects are masked due to observability issues and nature of the above-mentioned signature analysis, and *Formality* returns no suspicious nodes. The sixth column shows the number of suspicious gates that our method finds. Our method detects all of the Trojan circuit gates (no false negative gates) plus very small number of false positive nodes (benign gates). The seventh column shows the number of specification polynomials which is equal to number of flip-flops in the design plus number of primary outputs. The eighth column presents the number of implementation polynomials which is equal to number of fanout-free cones existing in the implementation. The CPU time (in seconds) to localize the Trojan is reported for each benchmark in ninth column. The time complexity of our method is linear with respect to the number of gates. The tenth, eleventh and twelve columns show the number of false positive gates that our approach, FACNI [134] and *Formality* [71] report, respectively. Clearly, our approach returns only few false positive gates. We are aware of the fact that comparison with FANCI is not fair since it does not requires golden model. However, FANCI returns a lot of suspicious gates that it may not include all of the Trojan gates. For example, FANCI has reported top twenty suspicious gates for S35932-T200, none of them are from Trojan gates. Moreover, FANCI returns a set of suspicious gates even when the circuit is Trojan free. The next columns show our improvement in comparison with *FANCI* and *Formality* based on number of false positive gates. Our approach has a significant improvement compared to

Table 6-2. The required tests to activate the Trojan

Benchmark	N=1			N=2			N=4		
	W/O Localization	With Localization	Improvement	W/O Localization	With Localization	Improvement	W/O Localization	With Localization	Improvement
RS232-T1000	311	13	23.9x	48205	78	618.0x	4E+8	715	5E+5x
RS232-T1100	310	14	22.1x	47895	91	526.3x	4E+8	1001	4E+5x
S15850-T100	2456	27	91.0x	3E+6	351	8.6E+3x	2E+12	17550	9E+7x
S38417-T200	5823	26	224.0x	2E+7	325	5.2E+4x	5E+13	14950	3E+9x
S35932-T200	5445	22	247.5x	1E+7	231	6.4E+4x	4E+13	7315	5E+9x
S38584-T100	7580	11	689.1x	3E+7	55	5.2E+5x	1E+14	330	4E+11x
Vga-lcd-T100	70162	22	3189.2x	2E+9	231	1.1E+7x	1E+18	7315	1E+14x
Average	13155.28	19.85	640.97x	2.9E+08	194.57	1.6E+6x	1.4E+17	7025.14	1.4E+13x

existing approaches - our approach reports orders-of-magnitude less false positive gates compared to [134] and [71].

6.2.3 Test Generation

For test generation, we used Tetramax [70], the ATPG tool from Synopsys to generate tests exhaustively to activate the reported suspicious nodes. Since our suspicious candidates are few, we can exhaustively check several combinations to activate the Trojan. However, without using our localization method or using heuristic methods such as [134], exhaustive method will not work due to large number of suspicious gates. Table 6-2 shows the number of tests needed for activation and detection of Trojans with/without using our localization method. First column shows the type of benchmark (same as Table I). The next two columns present the number of required tests to activate trigger conditions one at a time without and with using our localization method, respectively. The next column shows our improvement compared to without using localization. Our proposed approach improves number of required test vectors significantly. The next columns show the number of required tests to activate trigger conditions of two and four nodes at a time without and with using our localization method and the associated improvements, respectively. As it can be seen from Table 6-2, it is impractical to generate tests to activate four-node triggers even for these small benchmarks without our localization approach. If our localization is utilized, the number of required tests are reasonable and would be less by several orders of magnitude.

We also compared with MERO [30] for benchmarks S15850-T100 and S95932-T200. We did not compare using the remaining benchmarks because [30] did not report data for those benchmarks. Figure 6-4(a) shows the number of suspicious gates reported by our approach compared to MERO. Clearly, our approach provides up to 44 times (40 times on average) reduction in suspicious gates compared to MERO. Figure 6-4(b) compares the number of tests required to activate the Trojan. As shown in the figure, our approach requires up to two orders of magnitude (60 times on average) less test vectors compare to MERO.

The experimental results demonstrate four important aspects of our approach. First, the number of false positive gates are very small and in some cases there are no false positives. In these cases, our method is able to detect the whole Trojan circuit. Next, all of the Trojan payload and trigger gates are inside the list of suspicious gates. In other words, our approach does not produce any false negative result. Our approach detects both sequential and combinational Trojan circuits. Finally, our approach generates very few suspicious nodes (less than 0.2% of original design, less than 0.03% in most cases) that enables us to exhaustively generate tests to activate various trigger conditions to detect the Trojan circuit.

6.3 Summary

In this chapter, we presented an automated approach to localize functional Trojans in third-party IPs. First, we identified whether a third-party IP contains malicious functionality or it is trustworthy. Next, we presented an algorithm to localize the suspicious area of the Trojan-inserted IP to a region which contains very few (less than 0.03% of the original design in most cases) gates. Our approach does not require any unrolling or simulation of the design and it formally identifies the parts of the circuit that is Trojan free as well as the remaining suspicious gates. In order to further aid in Trojan detection, we proposed a greedy test generation method to activate the Trojan. Our experimental results demonstrated the effectiveness of the proposed methodology on trust-HUB benchmarks. Our localization approach reduces the overall Trojan detection effort (number of tests) by several orders of magnitude compared to the existing state-of-the art techniques.

CHAPTER 7
FSM INTEGRITY ANALYSIS IN CONTROLLER DESIGNS

Finite state machines (FSMs) control the functionality of the overall design. Any deviation from the specified FSM behavior can endanger the trustworthiness of the design. This is a critical concern when an FSM is responsible for controlling the usage or propagation of protected information (e.g. secret keys) in a secure component. FSM vulnerabilities can be created by a rogue designer or an attacker by inserting hardware Trojans in the FSM implementation. The vulnerability can also be introduced unintentionally by a CAD tool (e.g., when a synthesis tool is trying to optimize a gate-level netlist). In this chapter, we present an efficient formal analysis framework based on symbolic algebra to find FSM vulnerabilities. The proposed method tries to find inconsistencies between the specification and FSM implementation through manipulation of respective polynomials. Security properties (such as a safe transition to a protected state) are derived using specification polynomials and verified against implementation polynomials. In a case of a failure, the vulnerability is reported. While existing methods can verify legal transitions, our approach tries to solve the important and non-trivial problem of detecting illegal accesses to the design states (e.g., protected states).

There are limited efforts to identify and address the security vulnerabilities of a control circuit. Sunar et al. used Triple Module Redundancy (TMR) and parity checking methods to protect FSM of encryption algorithms against fault injection attacks [129]. However, the proposed technique introduces large area overhead (200%) and cannot detect other adversarial models such as hardware Trojans and vulnerabilities introduces by synthesis tools. In [136], a multilinear code selection algorithm is used to make cryptographic algorithm robust against fault injection attacks. However, this technique is not resilient against fault injection vulnerabilities caused by synthesis tools [44]. It has been shown that synthesis tools may insert additional *don't care* states in implementation of FSMs by using RTL don't care conditions and create assignments to optimize the gate-level netlist. At the same time, an adversary can use don't care states as a backdoor to access protected states and weaken the security of the overall design. In [44], authors use reachability as a trust metric to identify gate-level paths to protected states which do not exist in the RTL design. However, authors do not evaluate actual vulnerabilities caused by don't care states. They proposed an architectural change to state flip-flops in order to remove the access to the protected states from unprotected ones. Their proposed solution limits the functionality of the design. In [54], authors used mutation testing to detect existing hardware Trojans in unspecified functionality. However, mutation testing is very slow, and it may require significant manual intervention. Nahiyan et al. have proposed a state reachability analysis using ATPG tools [102]. They generate test patterns using the principle of n-detect-test [96] to extract the state transition graph (STG) of a given circuit. However, this option does provide any guarantees, e.g., in case one of their benchmarks they could not extract the whole STG. Sun et al. have proposed an FSM traversal technique using symbolic algebra [127]. However, their technique can only check the reachable states from a given state (e.g., initial state) and their technique cannot detect don't care states that may be introduced by synthesis tools. Similarly, they cannot detect hardware Trojans inserted in FSMs outputs.

In this chapter, we present a scalable formal approach that enables efficient FSM anomaly detection in state transition functions as well as FSM outputs. Our proposed method models the specification of a given FSM as a set of polynomials ($\mathbb{F}_{spec}$) such that each polynomial is responsible for describing all of the valid states that can be reached. Each output of the FSM also can be represented using one specification polynomial. The specification polynomials can be derived from RTL codes as well as design documents. We also partition the gate-level implementation of an FSM based on the boundary of flip-flops, primary inputs, primary outputs and fanout-free regions. We model each region by a polynomial and add it to the set of implementation polynomials ($\mathbb{F}_{imp}$). In the next step, we use Gröbner basis theory [40] to check the equivalence between two sets $\mathbb{F}_{spec}$

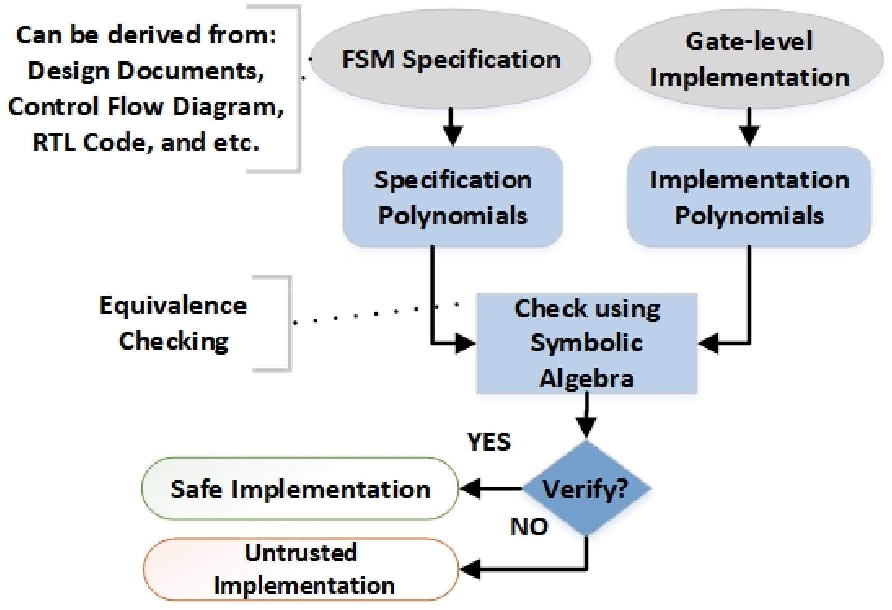

Can be derived from: Design Documents, Control Flow Diagram, RTL Code, and etc.
FSM Specification
Gate-level Implementation
Specification Polynomials
Implementation Polynomials
Equivalence Checking
Check using Symbolic Algebra
YES
Verify?
NO
Safe Implementation
Untrusted Implementation

Figure 7-1. Overview of FSM anomaly detection approach

and $\mathbb{F}_{imp}$. We reduce each specification polynomial F_{spec_i} using a set of implementation polynomials. If the reduction leads to a non-zero remainder, there are some vulnerabilities in implementation of F_{spec_i}. Every assignment that makes the remainder non-zero reveal the conditions that can activate the hidden malfunction.

Our approach is fully automated and it is guaranteed to find hard-to-detect FSM vulnerabilities in the implementation of an FSM when existing equivalence checking approaches fail. Experimental results demonstrate the effectiveness of our approach. Figure 8-2 shows the overall flow of our approach which the anomaly detection can be formally performed using our proposed equivalence checking method. We demonstrated the merit of our proposed method by detecting the vulnerabilities in various FSM designs, while state-of-the-approaches failed to identify the security flaws.

The rest of the chapter is organized as follows. Section 7.1 discusses the threat model. Section 7.2 illustrates our approach to detect FSM vulnerabilities. We show the effectiveness of our approach using the experimental results in Section 7.3 . Finally, conclusion is provided in Section 7.4.

7.1 Threat Model

In this section, we describe different categories of FSM vulnerabilities and show how an adversary can take advantage of these vulnerabilities to threaten the integrity of the overall design.

A state machine can be defined with six characteristics: an initial state S_{init}, set of possible states $\mathbb{S}$ where $S_{init} \in \mathbb{S}$, set of possible input events $\mathbb{I}$, a state transition function (F_T) that maps combination of states and inputs to states ($F_T : \mathbb{S} \times \mathbb{I} \rightarrow \mathbb{S}$), a set of output events ($\mathbb{O}$) and an output function (F_O) that maps states and inputs to outputs ($F_O : \mathbb{S} \times \mathbb{I} \rightarrow \mathbb{O}$). Based on the function F_T which defines transitions, each state S_i can be accessed through a set of immediate, authorized states as well as a set of specific input events. Set $\mathbb{A}_{S_i} = \{(S_j, I_j) | S_j \in \mathbb{S} \quad \& \quad I_j \in \mathbb{I}\}$ shows legal conditions to access state S_i and set $\mathbb{A}_{\mathbb{S}}$ shows all of the legal ways to access states $\mathbb{S}$. If state S_i can be accessed through the condition (S_m, I_m) where $(S_m, I_m) \notin \mathbb{A}_{S_i}$, it is a threat to the integrity of the design. In other words, state S_i should not be accessed through some illegal conditions/states which do not exist in the specification. From the security perspective, it is important that a design exactly performs as intended in the specification, nothing more nothing less. The extra access path to state S_i, (S_m, I_m) may endanger the integrity of the design as it may create a backdoor to access the critical secrets/assets. **In this chapter, we consider illegal access paths as threat model, and our goal is to identify them using symbolic algebra.**

The illegal access ways may be introduced by synthesis tools [44]. Behavioral specification (e.g. RTL) of an FSM may contain don't care conditions where the assignment to the next state or the next expected output is not defined (we call such FSMs incomplete FSMs). A synthesis tool takes an incomplete FSM and tries to assign deterministic values to the don't care conditions and transitions to generate an optimized circuit. As a result, a synthesis tool may introduce extra states and transitions to the gate-level implementation of the FSM which do not exist in the behavioral specification.

Formally, a synthesis tool may modify the set $\mathbb{A}_{\mathbb{S}}$ and convert it to $\mathbb{A}`_{\mathbb{S}}$. The extra set of access paths to the states of $\mathbb{S}$ can be computed as: $\mathbb{A}_M = \mathbb{A}`_{\mathbb{S}} - \mathbb{A}_{\mathbb{S}}$.

Set $\mathbb{A}_M$ (malicious access ways) can also be created/modified by a rogue designer or an attacker by inserting hardware Trojan in the FSM behavioral description as well as in the gate-level implementation of the FSM. The primary goal of the attacker is to create a backdoor to particular FSM states which may be triggered via an extremely rare input condition. The created backdoor may lead to a bypass of security protection of the design or create a denial of service. Moreover, malicious access ways can be set up by unintentional mistake of the designer. Example 1 illustrates potential threats in an FSM.

Example 1: The state transition diagram of a simple FSM is shown in Figure 7-2. The FSM has three states: G, C and protected state O representing with binary encoding 01, 10, and 00 respectively as shown in Figure 7-2. The FSM is responsible for checking a password before starting a specific operation. Operation state (O) should be accessed only from check password state (C) when a password is entered, and it is valid ($a = 1\&b = 1$). An adversary may use the unspecified conditions to insert illegal transitions to gain access to the operation state (protected state) from the state G without even entering the correct password to bypass the security protection ($a = 1\&b = 0$). On the other hand, the synthesis tool or the designer mistake can also introduce some unintentional illegal access ways (don't care states D) to the protected state and compromise the security of the design. With respect to the specification, $\mathbb{A}_O$ should be equal to: $\{(C, \text{“}a = 1\&b = 1\text{”})$. However, there are illegal access ways to state O in FSM implementation which is equal to: $\mathbb{A}_{M_O} = \{(D, \text{“}a\text{”}), (G, \text{“}a = 1\&b = 0\text{”})\}$. An adversary can compromise the security of the design by exploiting the existing vulnerabilities and attack the FSM. One of the possible attacks is fault injection attack [102]. The strategy is that the attacker tampers operating characteristics such as clock signal frequency, operating voltage or working temperature hoping to change different path delays and force the FSM to capture next state incorrectly. One example would be to force the FSM to go to the don't care states which have access to protected states or attack target states. For instance, an attacker can inject a fault during transition 01 $\rightarrow$ 10 ($G \rightarrow C$) to end up in don't care state 11 which has an immediate access to the protected state O and bypass password checking process in Example 1. The other possible attack is that the adversary inserts hardware Trojan by manipulating state transition graph in order to access certain states when a specific input event is triggered. In this case, the adversary is considered as an in-house rogue designer or an untrusted vendor/foundry. For instance, Example 1 shows that an adversary has inserted a Trojan that provides an illegal access way to state O from state G. The Trojan is typically hard-to-activate (from the unspecified design space) with negligible effect on the design constraints such as area and power to avoid detection from existing verification and debug flow. ■

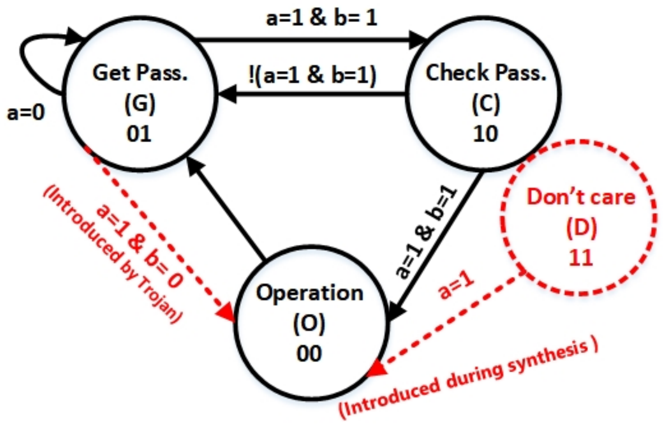


Figure 7-2. The state diagram for checking a password in order to perform a specific operation. Potential vulnerabilities are shown with dotted lines.

Based on above observations, any deviation of FSM implementation from the specification (including extra access ways) can endanger the overall design integrity. In the rest of this chapter, we propose a promising approach to analyze FSMs to find potential malicious functionality.

7.2 Finite State Machine anomaly detection

Although the presented approach of Chapter 4 is promising for verification of arithmetic circuit, applying it on a general sequential circuit is challenging due to several reasons. First, formulating the specification of a general circuit cannot be modeled as one simple and comprehensive polynomial. The specification may be modeled as a set of polynomials. However, finding the corresponding parts which are only responsible for implementing a special specification polynomial is not straightforward. Second, the implementation of a sequential circuit is not acyclic and it contains several loops which make the reduction operation infinite. Finally, time unrolling of the implementation is not efficient since it increases the design complexity and makes the equivalence checking inefficient. Moreover, existing Trojan may be activated after a large number of cycles (since the trigger condition is rare), therefore, there is no specific information about the required number of unrolling. In this chapter, we try to address the above-mentioned challenges to apply symbolic algebra to verify the trustworthiness of any general FSM. We not only check the given FSM for the correct expected behavior, but we also analyze the FSM to find any potential malicious extra access ways that may endanger the security of the FSM (nothing more). Finding extra access path especially from don't care states cannot be found using any formal methods such as model checkers since they are not accessed through the normal operation path. To the best of our knowledge, this is the first attempt in utilizing symbolic algebra in finding vulnerabilities in FSMs. The remainder of this section describes the different parts of our approach: deriving specification polynomials, generating implementation polynomials and performing equivalence checking in order to ensure the correctness of implementation and finding potential extra vulnerabilities.

7.2.1 Deriving Specification Polynomials

The specification of an FSM can be extracted from its state transition diagram or from a high-level description of the design (e.g., HDL modules). State transition graph can be derived from the design documentation as well as other high-level behavioral description of FSM such as RTL codes. In other words, deriving specification polynomials does not require a golden design/netlist.

Modeling the whole FSM using only one specification polynomial is not possible without considering the time notation in the specification polynomial as transitions between different states may be dependent on binary values of a specific input variable over different clock cycles. For example, as it is shown in Figure 7-2, state C can be accessed from path $G \rightarrow C$ when in two consecutive clock cycles t_1 and $t_1 + 1$ such that $a = 0$ in t_1 and $a = 1, b = 1$ in $t_1 + 1$. Writing these conditions as a polynomial (part of the overall specification polynomial) without considering the timing will lead to a zero polynomial as $(1 - a).a.b = 0$. However, if we add timing notations to our variables, the implementation also has to be time unrolled to match with the specification which increases the complexity of the equivalence checking problem. As a result, representing the functionality of an FSM using one specification polynomial is not possible. We propose an approach to model the specification of the FSM using polynomials without time unrolling the design.

Transitions of an FSM can be decomposed as: $F_T = \bigcup_{i=1}^{n} \mathbb{A}_{S_i}$ where n is the number of states and $\mathbb{A}_{S_i}$ shows all of the possible access ways of state S_i and F_T is the transition function of the FSM. To derive a set of specification polynomials which represent the whole FSM, we model each of $\mathbb{A}_{S_i}$ as one polynomial representing the legal access ways to state S_i and we add it to the set $\mathbb{F}_{spec}$.

A valid transition to state S_i happens when the current state is one of the authorized states and the corresponding input conditions are valid. In other words, S_i will be reached in the next clock cycle when the current state is S_j and condition $C_{j \rightarrow i}$ where $(S_j, C_{j \rightarrow i}) \in \mathbb{A}_{S_i}$ are evaluated to true. Note that, we show the value of variable x in the next cycle using x' notation. Therefore, transition $S_j \rightarrow S_i$ is modeled to a polynomial as: $f_{S_j \rightarrow S_i} : S_i' - (S_j.C_{j \rightarrow i}) = 0$. The polynomial of each of the conditions exits in $\mathbb{A}_{S_i}$ should be XORed to each other to derive a polynomial representing the whole $\mathbb{A}_{S_i}$ since only one of them should be valid at the same time. We illustrate our approach using Example 2.

Example 2: In order to extract specification polynomials for FSM shown in Figure 7-2, we consider each of the states independently and write a polynomial to represent conditions which update the next value of the state. For example, state O should only be accessed from state C when $a = 1$ and $b = 1$ or when the current state is state O and input a is equal to one. Since it should be accessed only from one of these conditions at a time, the conditions should be XORed to each other to show the effect of one condition at a time (the only exception is the condition of $a = 0$ in state G that will be ORed to other conditions since it works as the reset signal). The O' shows the next value of state O. The specification of the FSM shown in Figure 7-2 can be modeled as a set of three abstract polynomials ($\mathbb{F}_{spec} = \{f_G,\ f_C$ and $f_O\}$) as shown in Equation 7–1. ■

$$\begin{aligned}
&\mathbb{F}_{spec} : \{f_G : G' - ((1-a) \vee (C.(1-a.b) \oplus O)) = \\
&G' - (1 - a + a.O + 2.a.b.C.O + a.C - 2.a.C.O - 1.a.b.C) = 0 \\
&f_C : C' - a.b.G = 0 \\
&f_O : O' - (a.b.C) = 0\}
\end{aligned} \tag{7–1}$$

We will describe how specification polynomials are used to check security properties of an FSM in Section 7.2.3. Before performing the equivalence checking, we need to refine specification polynomials to apply proposed FSM equivalence checking process since the proposed method requires that specification variables' names be the same as the corresponding variables in the implementation. We refine specification polynomials based on the FSM encoding style as well as corresponding names of state flip-flops in the implementation (name mapping between flip-flop names and corresponding variables in specification polynomials). We refine the variables which represent states in specification polynomials based on naming and encoding information that can be found in the high-level description of the design such as RTL modules as we describe in Example 3. As a result, the specification of FSM outputs can also be modeled with word-level specification polynomials based on state variables as well as primary inputs.

Example 3: Suppose that the RTL code shown in Listing 1 is the RTL version of the state machine shown in Figure 7-2. We can see that states G, C and O are encoded as $\{01, 10, 00\}$ respectively. The state variable and next states are presented using variables $\{s_0, s_1\}$ and $\{n_0, n_1\}$. Therefore, the variables shown in Equation 7–1 can be updated based on the above-mentioned information. For instance, variable G and next state variable G' can be modeled as $(1 - n_1).n_0$ and $(1 - s_1).s_0$, respectively. As a result, the specification polynomials shown in Equation 7–1 can be rewritten as shown in Equation 7–2. Note that, considering C encoded as $s_1.(1 - s_0)$ and O as $(1 - s_1).(1 - s_0)$, the term $-2.C.O$ as well as $2.a.b.O.C$ of F_G in Equation 7–1 are evaluated in updated specification polynomials). ■

$$\begin{aligned}
&\mathbb{F}_{spec} : \{f_G : (1 - n_1).n_0 - (1 - a.b.s_1 + a.b.s_0.s_1 - a.s_0) = 0 \\
&f_C : n_1.(1 - n_0) - (a.b.(1 - s_1).s_0) = 0, \\
&f_O : (1 - n_1).(1 - n_0) - (a.b.s_1.(1 - s_0)) = 0\}
\end{aligned} \tag{7–2}$$

Specification polynomials can be extracted directly from the RTL modules by using some specific rules. The logical operations in *If* statements can be mapped to polynomials. For example, by considering the encoding, line $G : if(a == 1`b1\&\&b == 1`b1)n <= C$ can be modeled as equation $n_1.(1 - n_0) = a.b.(1 - s_1).s_0$ In the next step, the corresponding polynomials of *If Then Else* are XORed together to achieve the exclusive nature of these statements. The derived specification polynomials will be used in the equivalence checking procedure.

Listing 7.1. RTL module of FSM shown in Figure 7-2.

```
module fsm(input clock, a, b; output valid );
reg[1:0] s, n;
parameter O=2'b00, G=2'01, C=2'b10;
always @(a, b, s) begin
case(s)
  G: if(a == 1'b1 && b == 1'b1) begin
```

```
        n <= C;
    end else if (a == 0) begin
            n <= G; end
    C: if(a == 1'b1 && b=1'b1 )
            n <= O;
    else
        n <= G; end
    O: n <= G;
end
always @(posedge clock)
begin
    if(a==1'b0) s <= G;
    else s <= n; end
end
endmodule
```

7.2.2 Generation of Implementation Polynomials

Our goal is to partition the design and find the regions that are responsible for implementing each of the states and represent them as implementation polynomials. In order to perform this task, a mapping between state names and their corresponding gate-level state flip-flop names is needed. Here, we assume that the name of state inputs, outputs as well as state flip-flops are same between specification (RTL, state diagram, etc.) and implementation, or name mapping can be done based on existing methods in [100]. For the ease of the illustration, we explain how to extract the implementation polynomials when the FSM encoding is binary-encoding. Our proposed approach works for any state encoding.

After name mapping, we partition the gate-level implementation of the FSM based on state flip-flops. The state region construction starts from the input of the corresponding state flip-flop. The region construction continues with the inputs of the state flip-flop and moves backward recursively until it reaches to primary inputs or flip-flop outputs.

The constructed region is converted to a polynomial by converting each of its gates to a polynomial as shown in Equation 3–1 and combining them to each other to create one polynomial representing the whole region. We illustrate our approach using Example 4.

Example 4: Figure 7-3 shows the gate-level netlist which implements the FSM shown in Figure 7-2. In the implementation, FSM states are encoded using binary scheme (two flip-flops are used to implement the functionality of three states shown in the state diagrams of Figure 7-2). The implementation is partitioned starting from the input of state flip-flop n_i and it is continued until reaching either primary inputs or outputs of state flip-flops (s_i). In the next step, the corresponding polynomial of each partition is derived by combining polynomials of each gate in the region to represent the functionality of next state variables (n_i). The implementation polynomials are shown in Equation 7–3.

■

$$\begin{aligned}&\mathbb{F}_{imp} : \{n_0 - (1 - a.b.s_1 - a.s_0 + a.b.s_0.s_1) = 0,\\ &n_1 - (a.b.s_0 - a.b.s_0.s_1) = 0\}\end{aligned} \tag{7–3}$$

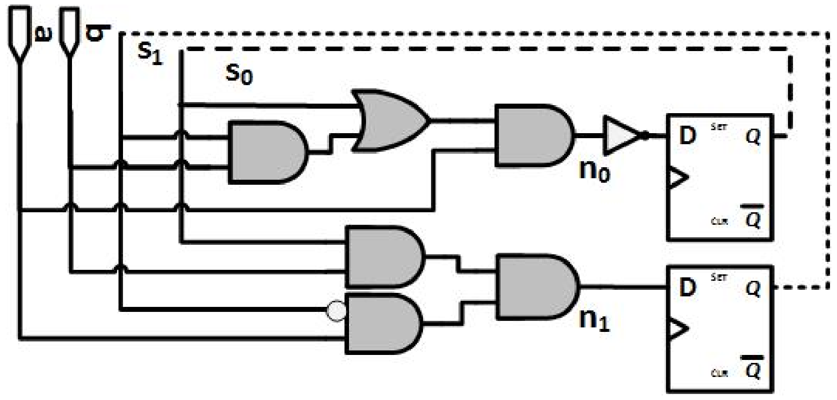


Figure 7-3. Implementation of FSM in Figure 7-2 using binary encoding.

When a gate's output goes to more than one gate, it is called a fanout. A fanout-free region is a set of gates that are directly connected together. Therefore, we partition the implementation to fanout-free regions and model each of them as one polynomial. The corresponding polynomials of each next state variable (n_is) can be computed by combining the polynomials of the corresponding fanout-free regions. Polynomials of fanout-free regions are calculated in order to reduce the efforts of implementation polynomial generation since one fanout-free region may be used in constructing the functionality of several n_is. Note that, in the implementation shown in Figure 7-3, the functionality of each n_i is constructed with only one fanout-free cone.

Note that, the implemented functionality of FSM's outputs also can be formulated as a function of FSM inputs and states and presented as polynomials. In order to find implementation polynomials corresponding to FSM's outputs, each output gate is considered and traversed backward until it reaches to either input/output of state flip-flops or FSM inputs. The traversed gates are modeled using one polynomial showing the functionality of the corresponding output, and those polynomials are added to set $\mathbb{F}_{imp}$.

7.2.3 Equivalence Checking

From the security point of view, it is important to make sure that the implementation of a design performs exactly its specification. We check the functional equivalence between a control logic specification and its implementation in order to establish the trust of the control logic. In this chapter, we formulate the FSM equivalence checking as ideal membership testing based on Gröbner Basis theory. Implementation polynomials $\mathbb{F}_{imp}$ are formed as an ideal I based on particular order $>$ (the topological order which exists in the implementation). FSM implementation is trustworthy if all of the specification polynomials in set $\mathbb{F}_{spec}$ are the member of ideal $I =< \mathbb{F}_{imp} >$.

In order to check the trustworthiness of the implementation, each specification polynomial F_{spec_i} from set $\mathbb{F}_{spec}$ is reduced over polynomials in $\mathbb{F}_{imp}$. All of the variables in specification polynomials (except primary inputs and flip-flops' outputs) are substituted with the corresponding functionality of the variable from the implementation polynomials. Note that, the reduction procedure is done using sequential polynomial division as shown in Section 3. The reduction process continues until a zero remainder or a non-zero polynomial which contains a combination of primary inputs and flip-flop outputs is reached. If reduction F_{spec_i} over set $\mathbb{F}_{imp}$ results in a zero remainder, it means that F_{spec_i} belongs to the ideal $I =< \mathbb{F}_{imp} >$. In other words, set $\mathbb{F}_{imp}$ has successfully implemented the specification F_{spec_i}. Otherwise, the implementation of F_{spec_i} is not trustworthy (implementation is not equal to specification). If all of the remainders are equal to zero polynomials, it means that the overall implementation is equal to FSM's specification since set $\mathbb{F}_{spec}$ includes specification of the FSM states as well as specification of FSM's outputs (specification polynomials cover all specification space). Algorithm 14 shows the equivalence checking procedure.

Algorithm 14: FSM Equivalence Checking Algorithm

```
1: **Input:** Gate-level netlist *imp* and specification polynomials $\mathbb{F}_{spec}$
2: **Output:** FSM anomalies $\mathbb{E}$
3: $\mathbb{F}_{imp}$=findImplementationPolynomials($imp$)
4: **for** each $f_{spec_i} \in \mathbb{F}_{spec}$ **do**
5:     $r_i$ = reduction of $f_{spec_i}$ over $F_j s \in \mathbb{F}_{imp}$
6:     **if** $(r_i! = 0)$ **then**
7:         $\mathbb{T}_i = findNonZeroAssignments(r_i)$
8:         $\mathbb{E}.put(f_{spec_i}, \mathbb{T}_i)$
9:     **end if**
10: **end for**
11: **Return:** $\mathbb{E}$
```

Algorithm 14 takes the gate-level netlist *imp* of a given FSM as well as the specification polynomials $\mathbb{F}_{spec}$ as inputs and tries to find any existing anomalies in the FSM. First, it computes the implementation polynomials ($\mathbb{F}_{imp}$) as described in Section 7.2.2 (line 4). In the next step, every specification polynomial f_{spec_i} (corresponding to state S_i) in $\mathbb{F}_{spec}$ is reduced over a set of implementation polynomials F_js using Gröbner Basis theory in order to find the remainder r_i (line 6). If the remainder is non-zero, it means that there are some malicious functionality in implementing specification polynomial f_{spec_i}. Every assignments that make the remainder non-zero, activates the malicious access path to S_i. The Algorithm stores the anomalies in the map $\mathbb{E}$ (lines 7-9).

Example 5: Consider the specification polynomials of Equation 7–2, gate-level netlist in Figure 7-3 as well as implementation polynomials shown in Equation 7–3. The

Equation 7–4 shows the equivalence checking procedure with respect to topological order $\{n_1, n_0\} > \{s_1, s_0, a, b\}$. Note that, reducing of variables $\{n_1, n_0\}$ happen at the same time as their orders are the same. However, we show the reduction of F_{spec_1} in two steps to illustrate the procedure better. ■

$$
\begin{aligned}
&F_{spec_1} : f_G : (1 - n_1).n_0 - (1 - a.b.s_1 + a.b.s_0.s_1 - a.s_0)\\
&stp_{11} : (1 - a.b.s_0 + a.b.s_0.s_1).n_0 - (1 - a.b.s_1 + a.b.s_0.s_1 - a.s_0)\\
&stp_{12} : (1 - a.s_0 - a.b.s_1 + a.b.s_0.s_1) - (1 - a.b.s_1 + a.b.s_0.s_1 - a.s_0) = 0\\
&F_{spec_2} : f_C : n_1.(1 - n_0) - (a.b.(1 - s_1).s_0)\\
&stp_{21} : (-a.b.s_0.s_1 + a.b.s0) - (a.b.(1 - s_1).s_0) = 0\\
&F_{spec_3} : f_O : (1 - n_1).(1 - n_0) - (a.b.s_1.(1 - s_0))\\
&stp_{31} : (a.s_0 - a.b.s_0 + a.b.s_1) - (a.b.s_1 - a.b.s_1.s_0) =\\
&(remainder) : a.s_0 - a.b.s_0 + a.b.s_0.s_1
\end{aligned}
\tag{7–4}
$$

As shown in Equation 7–4, specification polynomials of states G and C are reduced to zero which means that they are safely implemented by the gate-level netlist. However, the reduction of specification polynomial of the protected state O results in a non-zero remainder. The remainder reveals potential vulnerabilities in the gate-level implementation of the design to access the protected state O. Every assignment that makes the remainder non-zero, discloses an unauthorized access path to the state O. Table 7-1 shows the malicious access paths. As it can be observed from Table 7-1, don't care state $\{s_1, s_0\} = 2\text{`}b11$ can access the protected state O due to synthesis tool optimization (when input a is true). There is another malicious access path to the state O from state G when $a = 1$ and $b = 0$. This extra access is a hardware Trojan that was inserted by an adversary or a rogue designer.

Table 7-1. Malicious access paths to the protected state O shown in Figure 7-2

s_1	s_0	a	b
1	1	1	X
0	1	1	0

7.3 Experiments

7.3.1 Experimental Setup

In order to evaluate the effectiveness of our FSM anomaly detection approach, we have implemented the proposed Algorithms using Java. Our experiments were run on a PC with Intel core i7 and 16 GB memory. We have applied our method on various FSM benchmarks from *OpenCores* [66]. The benchmarks are described using RTL modules (that we treat as the specification). To obtain the gate-level implementation, we synthesize RTL modules using *Synopsys Design Compiler* [1]. We extract specification polynomials from RTL modules of FSM benchmarks considering their state transitions and output assignments. We have implemented a Java program such that we define the valid transitions to states in the form of abstracted polynomials and it generates one specification polynomial representing all of the logical transitions to a given state. The same approach was used to produce the specification polynomials for FSM outputs. On the other hand, implementation polynomials are driven automatically from the synthesized gate-level netlist using our proposed framework. In order to generate implementation polynomials, gate-level netlist is partitioned into the fanout-free regions which are restricted to flip-flops boundaries as well as primary input and primary outputs. We use fanout-free regions to reduce the number of implementation polynomials. We reduce specification polynomials over a set of implementation polynomials and each non-zero remainder represents an FSM security threat. The goal is to find the assignments to activate the vulnerabilities (if any).

7.3.2 Results

We have conducted two sets of experiments based on whether the vulnerability is introduced by the synthesis tool (unintentional) or an attacker (intentional). In the first set of experiments, the gate-level implementations are Trojan-free, and all the potential vulnerabilities are caused by the synthesis tool. Note that different encoding styles and values can create different vulnerabilities. In the second set of experiments,

Table 7-2. Result of the proposed FSM Anomaly detection technique using Equivalence Checking.

Benchmark	Encoding	#Gates	#FF	#Sts	#Trans.	Our Approach		
						DC Sts	DC Tran.	EQ (s)
TAP controller	One-Hot	136	16	16	33	3	6	80.63
AES Encryption	One-Hot	88	5	5	11	0	0	6.26
AES Encryption	Binary	60	3	5	11	3	6	5.03
RSA Encryption	One-Hot	114	7	7	9	0	0	18.48
RSA Encryption	Binary	76	3	7	9	1	1	6.2
SHA Digest	One-Hot	153	7	7	47	121	121	50.89
multiplier Controller	binary	52	3	5	8	3	3	1.85
SAP controller	Binary	135	4	12	25	0	0	17.23

we have inserted hardware Trojans in state transitions as well as state outputs of the implementations in order to show the effectiveness of our approach. The results are shown in Table 8.4.2 and Figure 8-4, respectively.

Table 8.4.2 represents the result of proposed FSM equivalence checking approach for eight different benchmarks. The first column shows the type of the benchmark. The second column represents the encoding style of the FSM design. We have considered binary and one-hot encoding methods to show that our proposed approach is not dependent on the encoding approach. The third, fourth and fifth columns represent the number of gates, number of state flip-flops, and the number of states, respectively. The sixth column represents the number of transitions in the FSM design. The next two columns indicate the number of don't care states and don't care transitions that our method finds, respectively. Note that our method does not report the don't care states that are not connected to any other states. Finally, the last column shows the CPU time that our proposed equivalence checking (EQ) approach to find anomalies in FSM benchmarks.

To show that our proposed approach can also detect hardware Trojans inserted in the state transition function as well as in the logic that generates the outputs of the FSM, we inserted hardware Trojans by exploiting the unspecified functionality of different benchmarks. Figure 8-4 shows the required time to detect the injected Trojan. The attributes of the benchmarks are the same as shown in the Table 8.4.2.

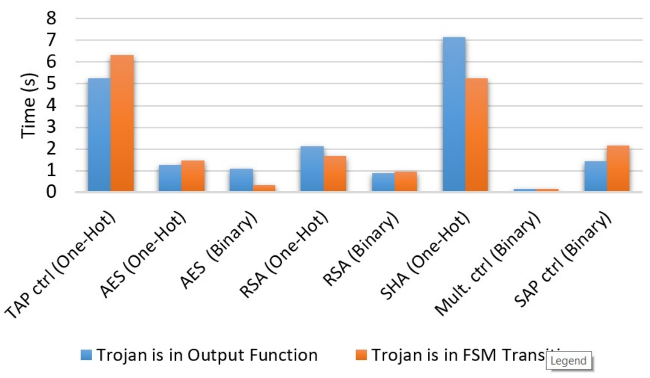


Figure 7-4. Time required to detect hardware Trojans in output logic and state transition function.

The experimental results demonstrated that our approach could detect the hidden vulnerabilities introduced by synthesis tool optimization while Formality fails to detect them. Note that some state encodings are more likely to have vulnerabilities caused by synthesis tools. For example, the synthesis tools tend to map all of the don't care states to a state with all zero's encoding (e.g. 3'b000) assuming that the state represents reset or ideal state. If the protected state is mapped using this encoding, there may be a direct access to the protected state from some don't care state caused by the synthesis tool.

7.4 Summary

It is critical to make sure that FSMs are correctly implemented, and there is no deviation from the specified functionality of the FSM since any unexpected functionality can endanger the integrity of the whole design. FSM vulnerabilities can be caused intentionally through an adversary by inserting hardware Trojan in the implementation or unintentionally using CAD tools such as synthesis tools. In this chapter, we presented an approach to formally detect anomalies in finite state machines using symbolic algebra. Our proposed approach models the specification of an FSM as a set of polynomials such that each polynomial represents all of the valid transitions to one of the states of the FSM. We modeled the implementation of an FSM as a set of polynomials. We check the equivalence of the specification polynomials and implementation polynomials using Gröbner basis theory. We have showed our approach can detect hidden vulnerabilities created by both synthesis tools or an adversary.

CHAPTER 8
TROJAN ACTIVATION BY INTERLEAVING CONCRETE SIMULATION AND SYMBOLIC EXECUTION

Modern System-on-Chip (SoC) designs consist of a wide variety of computation, communication and storage related Intellectual Property (IP) blocks. Developing and verifying each of these IP blocks in-house is infeasible due to time-to-market and budget constraints. It is a common trend in industry to rely on third-party IPs to keep the cost low and to meet firm deadlines. However, using IPs gathered from untrusted third-party vendors introduces security and trust concerns. These IPs may come with malicious modifications (hardware Trojans) inserted by a rogue designer or an adversary. Hardware Trojans can be hidden under rare branches or assignments such that they are triggered under extremely rare input sequences. As a result, traditional validation approaches are unable to activate them. A Trojan can leak secret information, create backdoor for attackers, alter functionality, degrade performance, halt the system, etc. [18, 81, 130, 131]. Therefore, it is crucial to have effective validation techniques to detect hardware Trojans.

Trojan detection methods based on side-channel analysis monitor changes in physical characteristics such as power, delay, and current [74, 103, 120, 137]. However, these approaches are unable to detect functional hardware Trojans since they mostly consist of a few gates which have a negligible effect on physical characteristics. Moreover, the minor change in side-channel signature may not be detectable due to relatively large environmental noise and process variations. The other class of methods relies on statistical parameters to distinguish Trojan-inserted circuits from Trojan-free circuits [135, 142]. However, these methods can lead to many false positives even if the circuit is Trojan-free. Logic testing based methods focus on functional comparison instead of looking at side-channel signatures. All these above-mentioned techniques require input vectors for activating the Trojan to measure the difference from expected behavior. Researchers have proposed model checking based approaches [73] for activating hardware Trojans. However, these methods suffer from inherent capacity restrictions of formal methods while dealing with large designs. Therefore, such methods are not effective for activating hardware Trojans in complex register-transfer language (RTL) models.

In this chapter, we propose a scalable directed test generation method to activate potential hardware Trojans in RTL designs using concolic testing. Concolic testing is an effective combination of concrete simulation and symbolic execution. It is scalable since it can avoid state space explosion by exploring one execution path at a time in contrast to dealing with all possible execution paths simultaneously (like conventional formal methods). While this approach has shown promising results in the context of software verification using concolic testing [59, 123], this area has not been explored in the context of test generation for detecting hardware Trojans. This chapter makes the following four important contributions.

- To the best of our knowledge, our proposed approach is the first attempt in developing an automated and scalable technique to generate directed tests to activate hardware Trojans in RTL models.
- We develop a threat model involving rare branches and rare assignments in RTL designs. This threat model leads to a list of potential security targets for directed test generation.
- We propose an effective combination of concrete simulation and symbolic execution to generate directed tests to activate these security targets.
- We show that detection of hardware Trojans boils down to coverage of rare branches and assignments in RTL models. Our experimental results demonstrate the effectiveness of our approach by activating hard-to-detect Trojans in large and complex benchmarks.

The remainder of the chapter is organized as follows. We provide a brief overview of concolic testing in Section 8.1. We present our threat model in Section 8.2. Section 8.3 describes our test generation framework for hardware Trojan detection. Section 8.4 presents our experimental results. Finally, Section 8.5 concludes the chapter.

8.1 Background: Concolic Testing

Concolic testing generates tests by effective combination of concrete simulation and symbolic execution. The idea was first demonstrated in software domain [59, 122, 123], and later applied on hardware designs [89, 108]. The first step involves concrete simulation with an initial set of input vectors. The execution path taken by the simulation can be decomposed into a set of constraints, referred as path constraints. Next step is to force the execution through an alternate route. This is done by selecting one of the unexplored branches in the execution path, negate corresponding constraint, and symbolically solve it using a constraint solver. If the solver comes up with a solution input set, then for that input execution will go through that alternate branch. If no solution is found, another branch is selected for negation. These steps are repeated until required target branch is reached. Concolic testing explores one path at a time, thus not prone to state explosion problem like model checkers that consider all design state space at a time. This advantage makes it an attractive choice for large designs.

Unlike random and constraint random test generation approaches, concolic testing tries to cover the design space uniformly. Although concolic testing is more beneficial than random test generation approaches in most of the time, this approach still suffers from long runtime due to its exhaustive branch selection methodology. Figure 8-1 shows three different test generation approaches. If we apply random strategy as shown in (a), there is no guarantee that target will be reached. If we use concolic testing search which uniformly prioritizes each branch like in (b), then it will eventually reach the target, but may take infeasibly long time. Note that the strategy used for alternate branch selection determines how quickly we reach the target. In this chapter, we propose a directed test generation approach (c) that utilizes the distance feedback to quickly reach the desired target.

8.2 Threat Model

An adversary (e.g., a rogue designer or an untrusted IP vendor) can insert hard-to-detect Trojans in the RTL design to affect the trust level of the design. To escape the detection

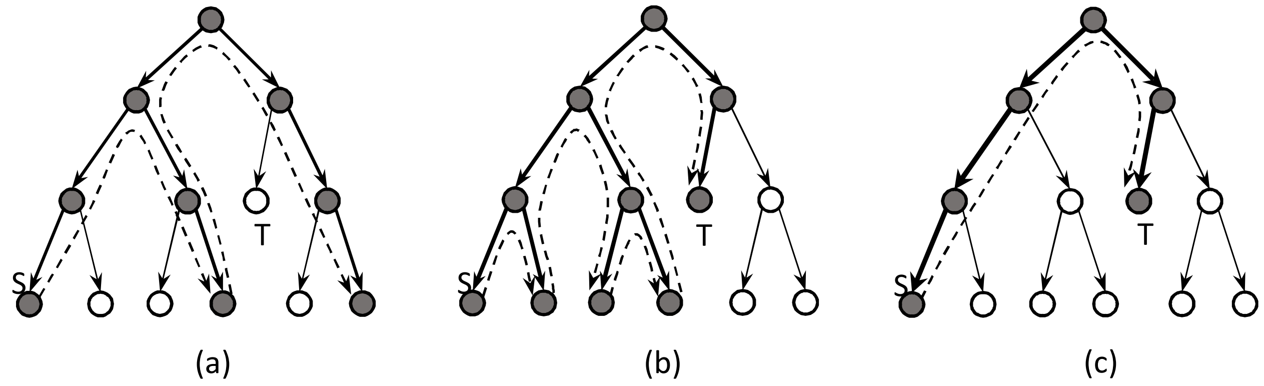


Figure 8-1. CFG traversal of different test generation methods. S is the start (current) node and T is the target. Covered nodes are dark colored. (a) Random tests: T is not covered. (b) Uniform tests using concolic testing: T is covered after many iterations. (c) Directed tests using proposed approach: T is quickly covered.

of the inserted Trojans during different steps of verification/validation procedure, Trojans are designed such that only a very rare set of input sequences can trigger them. In other words, Trojans are dormant during the normal execution, and activated under unusual (rare) conditions. Therefore, a smart adversary is likely to insert Trojans in RTL designs under rare branches which may reside in the unspecified functionality of the design. Otherwise, traditional simulation techniques using random or constrained-random tests can detect them, and the attacker's attempt would fail.

We also consider rare continuous/concurrent assignments in our threat model. These are assignments that may not be under any branches. Therefore, there is a high chance that they are not covered by targeting rare branch coverage. We transform rare assignments to branches without changing the functionality of the design in order to generate tests to cover them. Therefore, our threat model boils down to covering only rare branches including both original and newly created ones (due to conversion of rare assignments to branches).

Example 1: Suppose that an RTL IP contains an assignment statement which is as follows: $assign\ Tj\_Trig\ =\ count\_1\ \ \&\ \ count\_2$. Assuming Tj_Trig is the trigger signal of the hidden Trojan and it becomes true when both *count_1* and *count_2* overflow at the same time. Signal Tj_Trig becomes rare for value '1' if two of the counters are large enough. In order to consider such rare assignments in the test generation phase, we convert the assignment to a conditional statement as follows. ■

Listing 8.1. Converting an assign statement to a conditional statement.

```
if (count_1 & count_2)
        Tj_Trig <= 1;
else
        Tj_Trig <= 0;
```

In this chapter, we aim to activate the Trojans that change the functionality (e.g., causing information leakage or denial of service) and they can be triggered internally and externally. There may be cases when the Trojan's trigger is dependent on several rare branches. However, the trigger should be used somewhere in the design to cause the malicious functionality. Since the trigger value is rare, the branches/assignments that use the trigger would be rare to be activated, consequently. Therefore, by covering all of the rare branches and assignments, we can activate hidden Trojans in RTL designs.

While we use rare branches and rare assignments as our threat model in this chapter, our approach can easily incorporate suspicious nodes marked by other methods such as Transition Probability Calculator (TPC) [69], FANCI [135], VeriTrust[142] as well as score-based classification methods [105] to perform our Trojan detection analysis. Moreover, while this chapter uses Verilog examples, our approach is also applicable on VHDL designs.

8.3 Test Generation for Trojan Activation

Figure 8-2 shows the overview of our proposed approach. It consists of three major steps: i) design instrumentation, ii) obtaining security targets of the design based on identification of rare branches and assignments, and iii) directed test generation to activate security targets. The remainder of this section describes these steps in detail.

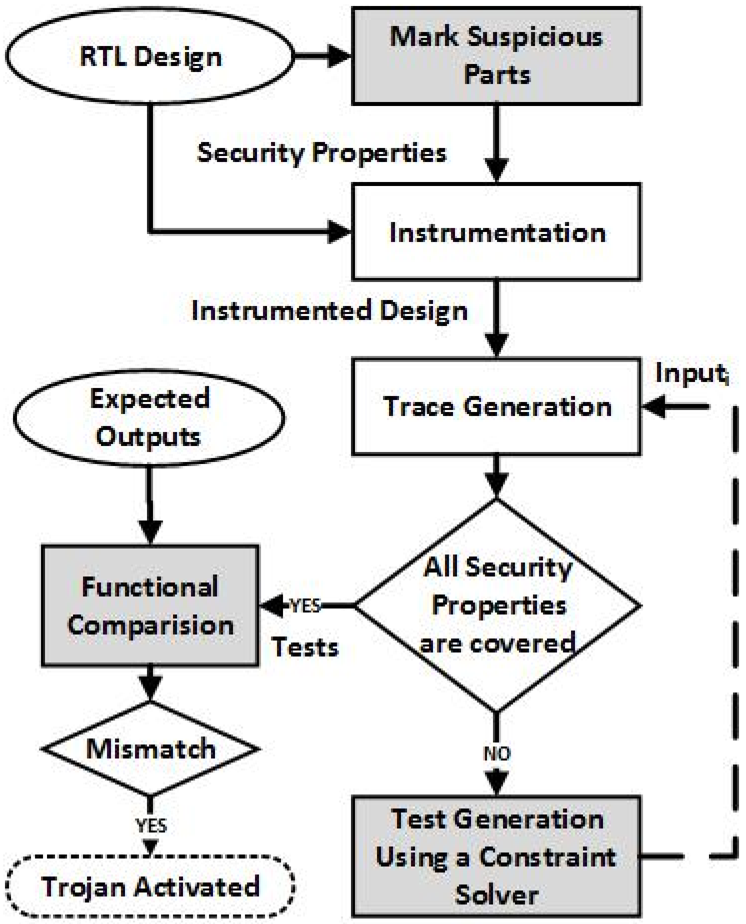


Figure 8-2. Proposed hardware Trojan activation framework consists of three main tasks: i) design instrumentation, ii) finding suspicious branches and assignments, and iii) test generation for Trojan detection.

8.3.1 Design Instrumentation

In order to both identify rare branches and generate path constraints during test generation, we need to generate the trace files from the simulation of the design. Therefore, we instrument the RTL code such that the sequence of events and the type of operations are recorded during different concrete execution paths. The design is flattened to ensure the uniqueness of variables during different clock cycles. Note that the instrumentation will not change the functionality of the design, since it only inserts print-related statements. The design is instrumented once, and traces are generated during the Trojan activation process.

Example 2: Listing 2 shows the instrumented version of Trojan circuit in AES-T100 benchmark [69] where "c" shows the current clock cycle (display statements are added). The Trojan is triggered when *state* variable becomes a particular value from 2^{128} possible values ($state = specific\_value$) [1] . ■

Listing 8.2. Instrumented version of a part of the code shown in AES-T100 [69]

```
always @(rst, state)
begin
        if (rst == 1) begin
                $display(IF rst==1 taken)
                $display((Tj_Trig, c+1) = 0)
                Tj_Trig <= 0;
        end else if (state==specific_value) begin
                $display(IF state==specific_value taken)
                $display((Tj_Trig, c+1) = 1)
                Tj_Trig <= 1;
        end
end
```

8.3.2 Identification of Suspicious Branches

To find rare branches which can potentially host hardware Trojans, we utilize random tests. We simulate the instrumented design using random tests. We count the number of times a branch is covered. We mark a set of branches as rare based on a specific threshold. For example, having a threshold of zero implies only uncovered branches as suspicious. Therefore, the default threshold should be zero. However, it can be a small number depending on the importance of specific IPs or designers' inputs. All the branches

[1] $specific\_value = 128`h00112233\_44556677\_8899aabb\_ccddeeff$

below the threshold are considered as security targets for the proposed Trojan activation framework.

Example 3: Consider the Trojan circuit shown in Listing 2, the *Trj_Trig* signal remains zero most of the time. However, when the *state* input gets the rare value shown in line 10, the *Trj_Trig* signal is activated. The chance of the branch shown in line 10 is being covered during random test simulation is extremely low (probability of $1/2^{128}$) and most likely it will not be covered. Therefore, our method marks this branch as a rare branch (security target). ■

After identifying rare branches, we model conditions of each rare branch as a security target such that the branch will be taken if the conditions are evaluated true. The security targets are used by our test generation framework to produce the input conditions (directed tests) to activate the respective rare branch in order to make sure that no Trojan or malfunction resides inside those rare branches.

Algorithm 15: Security Targets Identification

```
1: Input: Design under test DUT, Threshold $\pi$
2: Output: Set of security targets $\mathbb{P}$
3: $\mathbb{P} = \{\}, \Phi = \{\}$
4: $DUT'$ = InstrumentDesign(DUT)
5: for $i = 0; i < \pi; i++$ do
6:     Input vector $I$ = randomTest()
7:     Path Trace $\phi$ = Simulate($DUT', I$)
8:     $\Phi = \Phi \cup \phi$
9: end for
10: $A$ = identifyRareAssignments($DUT'$, $\Phi$)
11: $B$ = identifyRareBranches($DUT'$, $\Phi$)
12: for each $a \in A$ do
13:     $B = B \cup$ createEquivalentBranch(a)
14: end for
15: for each $b \in B$ do
16:     $P$=createSecurityTarget($b$)
17:     $\mathbb{P} = \mathbb{P} \cup P$
18: end for
19: Return $\mathbb{P}$
```

Algorithm 15 shows the procedure to mark security targets. The algorithm takes the design under test (DUT) as well as threshold π (indicates the number of random simulations as inputs) and it produces a set of targets $\mathbb{P}$ as output. To trace the execution path, the design is instrumented (line 4). Then, the design is simulated using random tests for π times, and simulation traces are stored in set Φ (lines 5-9). When the simulation phase is over, branches that are not covered are identified using the information extracted from simulation traces Φ and added to set B (lines 11). Note that for each branch b, two conditions are considered: i) b is taken, and ii) b is not-taken. The same procedure is done for identifying the rare assignments (line 10). Each rare assignment a is converted to an equivalent branch and added to set B (lines 12-14). Finally, for each branch b where $b \in B$ is converted to an assertion, and it is added to the output set $\mathbb{P}$ (lines 15-19).

Example 4: Consider the Trojan circuit shown in Listing 2. The branch shown in line 7, $if\ (state == specific\_value)$ has been not covered during random simulation, and thus it is marked as a rare branch. Therefore property $assert\ eventually\ state = specific\_value$ is added to the design for security validation.

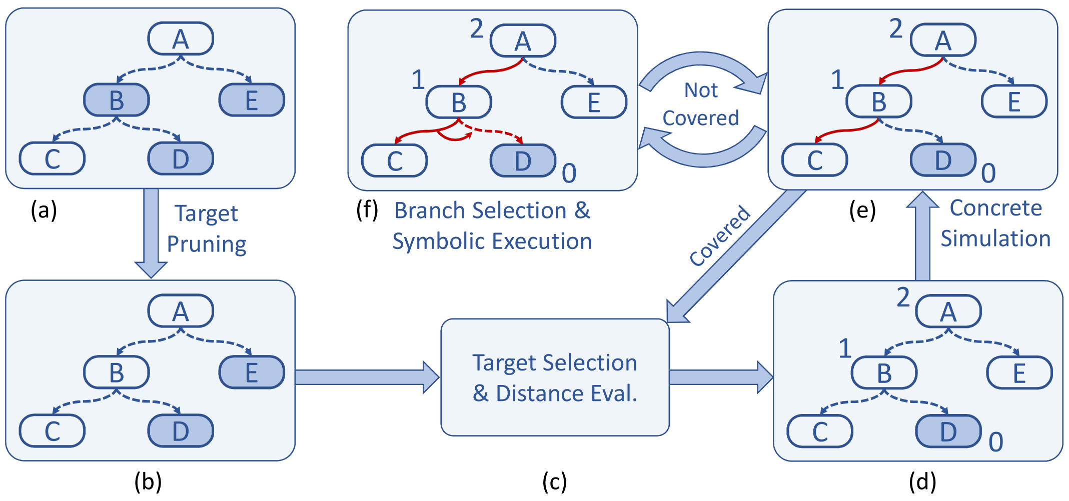
Similarly in Listing 1, since the condition $(count\_1 == 1 \quad \& \quad count\_2 == 1)$ is a rare event, we create a security target as: $assert\ eventually\ count\_1 == 1 \quad \& \quad count\_2 == 1$.

■

8.3.3 Coverage Guided Test Generation for Trojan Activation

Algorithm 16 takes an RTL design as well as security targets as inputs and generates directed tests to cover targets. First, we perform a preprocessing step to reduce the total number of security targets. The number of security targets has a direct impact on the performance of the test generation approach. The number of targets can be reduced based on the dependency between them due to the fact that all branches within a rare branch are also rare. Covering the inside branch will also cover the parent branch, and thus it can be removed from target list. Such dependency can be resolved by looking at the control flow graph (CFG) of the design. If a target is dominator of any other target, it can be pruned. An example is shown in Figure 8-3. Here (a) shows the initial targets as B, D, and E. However, B is a dominator of target D, hence can be removed. This is done statically, without unrolling the design for multiple cycle. Static analysis only prunes part of the dependent branches. Dynamic pruning with actual unrolling of design would result into more pruned targets, but we do not use it in this work since it is susceptible to state explosion.

After pruning step (line 4), one of the targets is selected for test generation. Distance from the target is then evaluated by running breadth first search (BFS) starting from the target branch, and following predecessor edges in the CFG. An example is shown in Figure 8-3(d). Here, D is selected to be covered first. Initially, target D is assigned distance 0 and all other branches are assigned infinity. Next, we run BFS starting from D, and follow predecessor edge. After distance evaluation is finished, distance would be: $B = 1$, $A = 2$ and others infinity. This procedure is also done statically without actually unrolling the design. Next, we apply concrete simulation followed by symbolic execution for several iterations in order to generate tests to activate the potential hardware Trojan. In each iteration, the instrumented design is simulated for a specific number of clock cycles and a trace file is produced (Figure 8-3(e)). The information of the trace file is then converted into path constraints (line 15). These constraints model the execution path taken by the concrete simulation. In the next step, one of the alternate branches is selected to be explored. We have selected the branch which has lowest assigned distance value (line 17). In other words, we have given priority to the branches that are closer to our security target. Path constraints that leads to that branch is then symbolically solved by a constraint solver (line 19). If a solution exists, then we again do concrete simulation with that solution, this time forcing execution through that alternate branch. This concrete and symbolic execution steps are repeated until target is covered (Figure 8-3(e)-(f)), or some terminating conditions are met (e.g. timeout). If all of the branches are exhausted and no new input vector I can be generated, algorithm returns generated tests (line 25).



A
B
E
C
D
(a)
Target Pruning
(b)
Target Selection & Distance Eval.
(c)
(d)
2
1
0
Concrete Simulation
(e)
Covered
Not Covered
(f) Branch Selection & Symbolic Execution

Figure 8-3. Overview of test generation procedure. Targets are darkened. (a) Initial targets. (b) Targets after pruning. (c), (d) Selects one target, and evaluates distance for that target. (e) Runs concrete simulation. Execution path is marked as red. (f) Selects an alternate branch and symbolically solves for input.

Example 5: Consider the instrumented code in Listing 2. The concolic testing generates a test vector to activate the rare branch (line 7) where $rst = 0$ and $state = specific\_value$. This input vector makes Tj_Trig true, and therefore, activates the Trojan. ■

We effectively utilize the advantages of both concrete simulation and symbolic execution to generate directed tests to activate hidden Trojans in the design. Our approach avoids the state space explosion by examining one path at a time in contrast to traditional formal methods that consider all simultaneously. Therefore, it is capable of reaching to hard-to-detect paths and activating hidden Trojans.

8.4 Experiments

8.4.1 Experimental Setup

Experiments are performed using a 64-bit Red Hat Enterprise Linux server machine with Core-i5 3427U CPU and 16GB of RAM. The CPU has two cores running at 1.80GHz. Our hardware Trojan detection approach is implemented using C++. Icarus Verilog is used for parsing and simulating the RTL design [139]. Yices is used as the constraint solver [45]. EBMC model checker is used to perform property-based test generation [86]. Our Trojan detection framework is tested with Trojan inserted designs of Trust-Hub

Algorithm 16: Generation Tests for Trojan Activation

```
1:  Input: Instrumented Design under test $DUT'$, Security targets $\mathbb{P}$
2:  Output: Set of test vectors $\mathbb{T}$
3:  $\mathbb{T} = \{\}$
4:  $\mathbb{P}' =$ pruneOverlappingTargets($\mathbb{P}$)
5:  Input vector $I =$ random()
6:  while  $I$ is not null do
7:      $\mathbb{T} = \mathbb{T} \cup I$
8:      Path Trace $\phi =$ Simulate($DUT', I$)
9:      for each $P \in \mathbb{P}'$ and isCovered($P, \phi$) do
10:         $\mathbb{P}'$.remove($P$)
11:     end for
12:     if $\mathbb{P}$ is empty then
13:         Return $\mathbb{T}$ {All security targets are covered}
14:     end if
15:     $C =$ findConstraints($DUT', \phi$)
16:     for all uncovered branches $C$ do
17:         $b =$ branchWithLeastDistanceFromTarget($C$, $p$)
18:         $b_n = \neg b$
19:         $I =$ satisfy($C + b_n$)
20:         if $I != null$  then
21:             break {To execute new input}
22:         end if
23:     end for
24: end while
25: Return $\mathbb{T}$
```

benchmark suite [69]. Additionally, some custom designed benchmarks are used to demonstrate the scalability of our method. Synopsys Design Compiler is used for synthesizing benchmarks. To identify suspicious branches/assignments, we simulated each of the design for one million clock cycles using random test vectors.

8.4.2 Results

In this section, we show the efficiency of our framework to generate test vectors to activate the hidden Trojans in Trust-Hub benchmarks. We compare our approach with model checking. Please note that there are no existing approaches for rare branch activation using model checking for Trojan detection in RTL models. We provided these results to show that EBMC model checker works well when the design is small. This also highlights the fact that our fault model and security targets can be utilized by existing approaches.

8.4.2.1 Results using Model Checking

The model checker takes the design and the negation of the security targets and tries to generate a counterexample (test) to cover the security targets. Table 8-1 shows the results of hardware Trojan detection using EBMC model checker. The first column indicates the type of the benchmark. The second column indicates the number of cycles that the design needs to be unrolled until the Trojan is activated. The third column shows the number of rare branches. The fourth column shows the coverage of the security targets. Finally, the last column presents the required time for the model checker to generate the tests to activate the Trojan. Based on Table 8-1, when the designs are small, and the security targets are not complex (are not dependent on too many internal states and variables), using EBMC is effective. However, using EBMC for large designs with complex targets is not effective as demonstrated in the next section.

Table 8-1. Results to generate directed tests that activate the Trojan using EBMC model checker in RTL Trust-HUB Benchmarks. NA= not applicable

Benchmark	Cycles Unrolled	#Rare Branches	Rare Branch Coverage	EBMC (sec)
RS232-T100	250	0	NA (*)	NA
RS232-T200(**)	250	1	100.00%	1.56
RS232-T400	250	2	100.00%	2.83
RS232-T800	250	1	100.00%	1.84
wb_conmax-T200	10	1	100.00%	8.71
wb_conmax-T300	10	1	100.00%	11.77

* All branches are covered during random tests.

** Corrected trigger condition. Original condition is impossible to trigger.

8.4.2.2 Results using Concrete Simulation and Symbolic Execution

Table 8-2 presents the results of Trojan detection using our approach on Trust-hub benchmarks. The first and second columns show the type of the benchmark and the number of unrolled clock cycles, respectively. The third column shows the number of security targets that are used for Trojan detection. The fourth column indicates the coverage of rare branches using our approach. The fifth and seventh columns present

Table 8-2. The required time and memory to generate directed tests that activate the Trojan using EBMC as well as our approach in RTL Trust-HUB Benchmarks. MO = memory out of 16 GB.

Benchmark	Cycles Unrolled	#Rare Branches	Rare Branches Coverage	EBMC		Our Approach		Time Improvement	Memory Improvement
				Time (sec)	Memory (MB)	Time (sec)	Mem (MB)		
wb_conmax-T200	10	1	100.00%	8.71	659.5	13.36	124.7	-1.53x	5.29x
wb_conmax-T300	10	1	100.00%	11.77	1198.9	11.06	118.8	1.06x	10.09x
AES-T500	10	5	100.00%	67.07	7436	11.67	599	5.74x	12.41x
AES-T1000	10	2	100.00%	68.37	7441	3.88	525	17.62x	14.17x
AES-T1100	10	5	100.00%	71.03	7449	11.8	601	6.01x	12.39x
AES-T1300	10	9	100.00%	68.57	7449	2.65	524	25.87x	14.21x
AES-T2000	10	6	83.33%	69.27	7554	6.75	600	10.26x	12.59x
cb_aes_01	5	1	100.00%	1.27	179.4	0.51	55.3	2.49x	3.24x
cb_aes_05	10	1	100.00%	11.47	1450.3	4.03	244.3	2.84x	5.93x
cb_aes_10	15	1	100.00%	33.17	4130.6	14.47	502.4	2.29x	8.22x
cb_aes_15	20	1	100.00%	70.78	8041.2	32.14	778.2	2.20x	10.33x
cb_aes_20	25	1	100.00%	110.13	13202.8	86.03	1085.5	1.28x	12.16x
cb_aes_25	30	1	100.00%	-	MO	150.54	1405.3	-	-
cb_aes_30	35	1	100.00%	-	MO	243.02	1780.3	-	-
cb_aes_35	40	1	100.00%	-	MO	371.23	2112.7	-	-
cb_aes_40	45	1	100.00%	-	MO	851.25	2532	-	-

the required time for Trojan activation using EBMC model checker and our approach, respectively. The sixth and eight columns show the required memory to generate the test to activate the Trojan using EBMC and our approach, respectively. The last two columns demonstrate the improvement over EBMC regarding the required time and memory.
The results in the first two rows of the table demonstrate that even though that EBMC may have comparable runtime results for small benchmarks, the memory requirements are significantly higher than proposed approach. Our approach has significantly (an order-of-magnitude) better runtime as well as memory results for larger benchmarks in comparison with EBMC. Besides Trust-Hub benchmarks, one additional custom benchmark is used to demonstrate scalability of our approach. The benchmark is named cb_aes_xx. It is a modified version of AES benchmarks, where the Trojan trigger depends on the xx-th round's output. As shown in Table 8-2, the bounded model checking tool EBMC fails to generate a test case due to state space explosion (out-of-memory error) while our approach works. This shows that the proposed approach can scale with design size. Table 8-2 demonstrates the significant reduction of the required memory using the proposed approach.

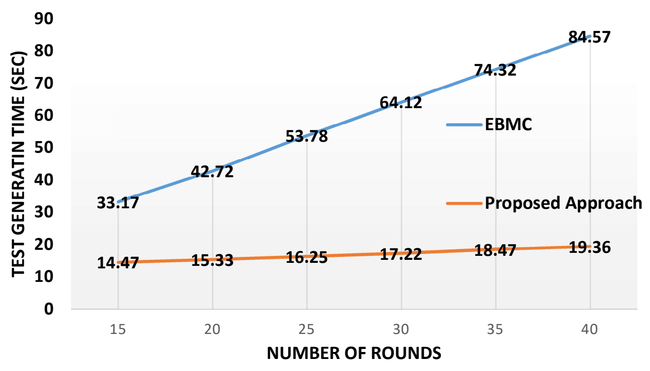


Figure 8-4. Comparison of test generation time to activate the Trojan in *cb_aes_xx* benchmarks.

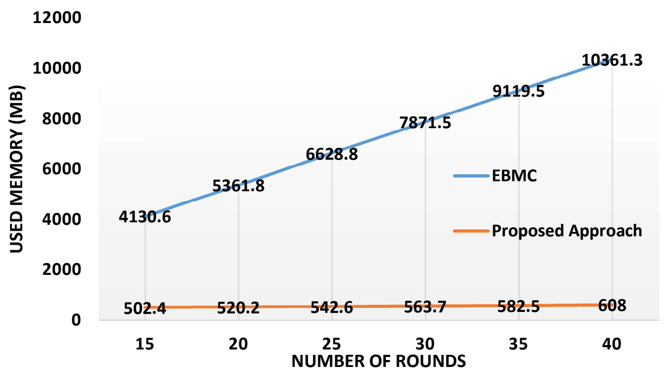


Figure 8-5. Comparison of memory requirement to activate the Trojan in *cb_aes_xx* benchmarks.

Figures 8-4 and 8-5 show the scalability of our approach in comparison with the model checking based approach for cb_aes_xx benchmarks when we change the number of AES encryption rounds. These rounds are cascaded sequentially, and the Trojan is designed such that its activation is dependent on all intermediate stages. Therefore, increasing the number of rounds lead to increasing the complexity of Trojan activation. Furthermore, as the Trojan activation depends on the last round, the design should be unrolled for at least the number of rounds. Figure 8-4 shows the required test generation time, and Figure 8-5 shows the memory requirement by the two approaches.

Our results demonstrate that effective interleaving of concrete and symbolic execution leads to a scalable approach for automated generation of directed tests to activate hard-to-detect Trojans in large RTL designs. The results show that the run time of our approach is comparable to model checking for small designs, but an order-of-magnitude faster for large designs. Moreover, the memory usage of our approach is an order-of-magnitude better than model checkers. As a result, our proposed approach can generate efficient tests to detect hidden Trojans when state-of-the-art approaches fail.

8.5 Summary

In this chapter, we presented an automated and scalable approach to activate hard-to-detect hardware Trojans in RTL designs. The first step in our detection methodology involves marking branches and assignments which are likely to contain Trojans. We used rarity of a(n) branch/assignment to make this decision. We proposed a threat model involving rare branches and rare assignments. We automatically generate security targets based on the fault model. We effectively utilize interleaved concrete simulation and symbolic execution to generate directed tests by covering security targets to detect potential hardware Trojan in the design. Our experimental results demonstrated that our test generation technique is scalable and effective for detecting Trojans in large RTL IPs when state-of-the-art methods fail. Our approach can be easily integrated into existing design flows and it can be applied on any RTL designs with single or multiple clock domains. Moreover, the proposed method can detect a wide variety of combinational and sequential Trojans in modern IP cores.

CHAPTER 9
OBSERVABILITY-AWARE POST-SILICON TEST GENERATION

A major challenge in post-silicon debug is to generate efficient tests that both activate requisite coverage goals on the target hardware as well as produce results that are observable through a given on-chip design-for-debug architecture. Unfortunately, such tests cannot be generated directly from RTL models, due to both design complexity and bugs in the design itself.

In this chapter, we present a technique for observability-aware post-silicon directed test generation through analysis of pre-silicon design collaterals. Our key approach to overcome the scalability and relevance challenges mentioned above is to exploit more abstract transaction-level models (TLM) for the designs to perform our analysis. TLM definitions are much more abstract, structured, and compact, compared to RTL, which permits effective application of exploration to identify high-quality directed tests. A key challenge is to map design functionality and observability between TLM and RTL so that the tests generated at TLM can be translated to effective, observability-aware tests for RTL. Our approach involves mapping test and observability requirements between TLM and RTL, enabling TLM analysis to generate post-silicon tests. We provide case studies to demonstrate the flexibility and effectiveness of the approach. In this dissertation, we propose an approach to address this problem by exploiting transaction-level models (TLM). We provide case studies from a number of different design classes to demonstrate the flexibility and generality of our approach.

The remainder of the chapter is organized as follows. Section 9.1 discusses our overall framework, some of the challenges faced, and our approach to overcome them. Section 9.2 discusses our experimental results. Finally, Section 9.3 concludes the chapter.

9.1 Observability-aware Test Generation

Suppose we have an RTL model M, a set of checkers and coverage conditions $\mathcal{A}$ to be exercised in post-silicon, and a set of traceable signals $\mathcal{S}$. Our goal is to develop directed tests for exercising $\mathcal{A}$ such that the results of the test can be inferred by observing the signals in $\mathcal{S}$. Our approach uses TLM models for post-silicon test generation. We impose some key constraints on the underlying design for viability of our approach. Our key requirement is the existence of a TLM description of the system (in addition to RTL), where the TLM is assumed to be the "golden" specification. Our second requirement is that TLM and RTL models must have the same external (input-output) interfaces. Since SoC designs re composed of a number of hardware or software intellectual property (IP) blocks, we require the IPs to have the same interface variable definitions in TLM and RTL models. Finally, we will only consider assertions from $\mathcal{A}$ that are stutter-insensitive. The general class of stutter-insensitive properties is LTL\X properties (X denotes next operator). Tests for stutter-insensitive properties are natural targets for generation based on TLM since the TLM models are untimed.

Fig. 9-1 provides an overview of our approach. The approach involves four important steps: i) mapping observability constraints as part of test targets, ii) mapping test targets from RTL to TLM, iii) test generation using TLM description, and iv) translating TLM tests to RTL. The basic idea is to transform a RTL assertion (ϕ) as well as observability constraints (ψ) to create a modified RTL assertion with observability constraints (π). The modified assertion needs to be mapped to TLM assertion (α). The TLM assertion/property would be used to construct a TLM test. Finally the TLM test would be translated to an RTL test. In the remainder of this section, we describe each of the steps in detail.

9.1.1 Mapping Observability Constraints

Let M_R and M_T be the RTL and TLM models of a design with (common) primary inputs $I = \langle I_1, I_2, ..., I_n \rangle$ and primary outputs $O = \langle O_1, O_2, ..., O_m \rangle$. Let $R = \langle R_1, R_2, ..., R_l \rangle$ be the set of observable RTL signals. Consider a stutter-insensitive RTL assertion ϕ over I, O and R from $\mathcal{A}$. For the purpose of the discussion below,

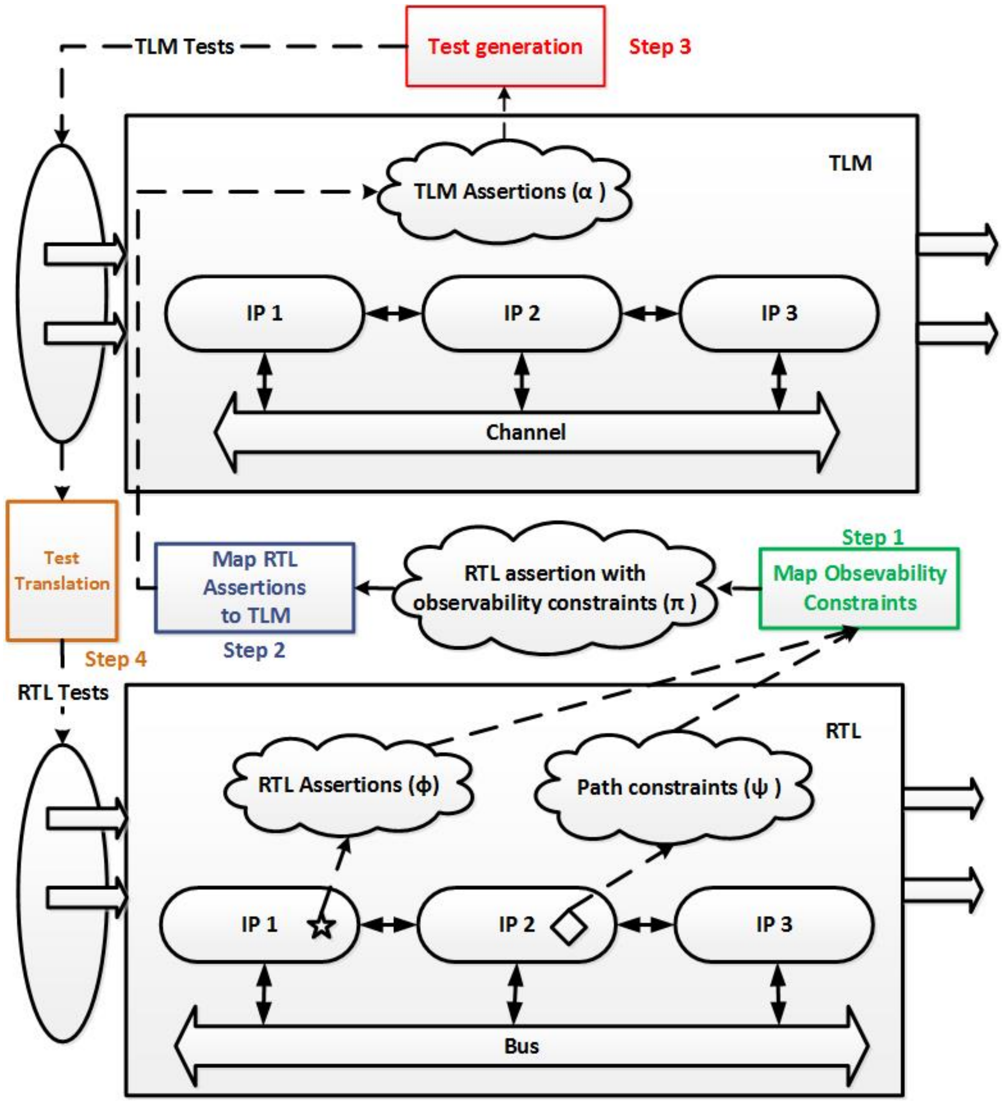


Figure 9-1. The proposed methodology with four important steps

it is convenient to think of ϕ as an LTL\ X formula. Our method for generating observability-aware tests for ϕ involves the following steps.

1. **Trace Cone-of-influence Calculation:** We traverse the control/data flow of the RTL backwards from the signals in R to the variables in ϕ. This cone-of-influence calculation is made under the constraint that ϕ holds. All signals in R whose cone-of-influence does not include any variable in ϕ are discarded.
2. **Assertion Propagation:** We use symbolic simulation to forward-propagate variables in ϕ along the cone of influence found in Step 1. The result is a restatement

of ϕ into a new formula ψ stated in terms of traceable variables (including signals in R and O).

3. **Assertion Abstraction:** We construct a formula π subsuming ϕ and ψ as follows. (i) If ψ is consequent of ϕ, $\pi : (\phi \rightarrow F\psi)$ and vice versa. (ii) If ϕ and ψ can be satisfied concurrently, $\pi : (\phi \wedge \psi)$.

Example 1: Figure 9-2 shows a router in RTL and TLM 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: i) parity ($data\_in[0]$ in RTL and $pkt\_in.parity$ in TLM) ii) payload ($data\_in[7..3]$ in RTL and $pkt\_in.payload$ in TLM) and iii) address ($data\_in[2..1]$ in RTL and $pkt\_in.addr$ in TLM).

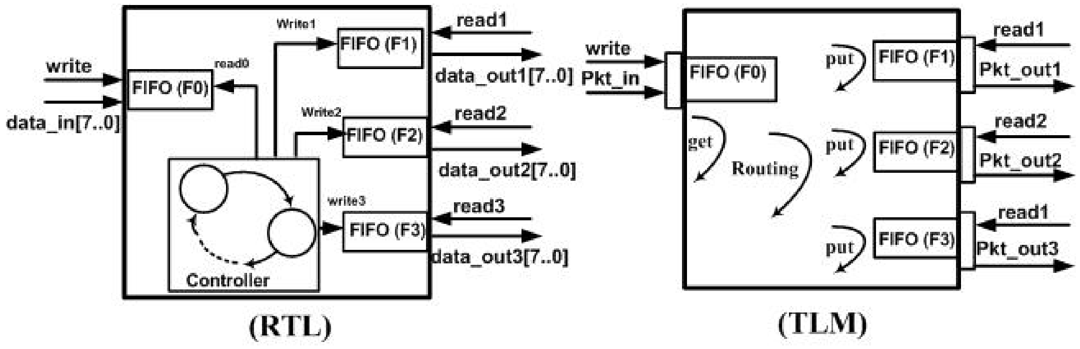


Figure 9-2. Router design, RTL and TLM implementations

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). Consider generating a test to check that signal $read0$ from F_0 (which is internal signal) is not stuck at zero. The corresponding assertion, written as an (LTL\ X) formula, is (ϕ : F $read0$). Suppose that the address part of input data of F_1 ($F_1.data\_in[2..1]$) is selected as a trace signal. In order to observe activation's effect of ϕ through $F_1.data\_in[2..1]$, the following predicate must be true two cycles after

$read0$ becomes true: $\psi : F_1.write1 \wedge F1.pkt\_in[2..1] = 1$. Thus, following the above steps we get:

$\pi : Fread0 \rightarrow XX(F_1.write1 \wedge F1.data\_in[2..1] = 1)$

9.1.2 Mapping Test Targets (Assertions) from RTL to TLM

The key challenge in mapping assertions from RTL to TLM is to bridge the abstraction level between the two designs. We achieve this by exploiting the commonality of interfaces. Our goal is to find TLM property α that is *test equivalent* of RTL assertion π constructed in the previous section. Here by *test equivalence* we mean that they generate equivalent tests or counterexamples. The problem reduces to transforming π into a formula α such that (i) α is an LTL\X property over I and O, and (ii) If a test T is a counter-example of α in TLM then T is also a counter-example to π in RTL. If π contains internal RTL variables, we need to turn it to a test equivalent RTL LTL\X property where it is only over the variables in the interface.

We can define α through symbolic simulation of variables in π analogous to the previous section, but this time over the TLM model. Suppose that π is a temporal logic formula over $P_1, P_2, \ldots, P_n$ where each P_i shows one condition on interface or internal variables. For propagation to interface variables, CDFG is traversed backward from point/points that P_i is true to reach primary inputs. [1] As a result, each of P_i be restated as a temporal formula θ over $Q_i = q_1, \ldots, q_m$ where q_j denotes a condition on interface variables. The next step is to remove exact timing notation from θ. Our approach is based on the observation that the original assertion ϕ is stutter-insensitive. Thus distributive property is applied such that their operands are atomic (e.g. $X(q_i \wedge q_j) \equiv X(q_i) \wedge X(q_j)$). In addition, we apply the following rules.

[1] For some assertions like $P_i \quad \rightarrow P_j$, backward traversal from P_i and forward traversal from P_j would be beneficial. However, in most of the cases performing one of them is enough.

- $F\ (Xp) = F\ p$. Thus, operator F can subsume X.
- A property $p \rightarrow XX...Xq$ can be replaced by $p \rightarrow Fq$.
- A property $p \wedge X\neg p$ can be replaced by $p \wedge F\neg p$.
- A property $p_1 \wedge X..Xp_2$ can be replaced by $p_1 \rightarrow F(p_2)$ when p_2 is a condition on variables from set O.

The modified assertion is an LTL\X that contains conditions on interface variables so it can be applied on TLM. In fact, assertion π is mapped as a sequence of *put* and *get* transactions. The next step is to perform name mapping when the interface signal names are not identical. The resultant assertion is our desired assertion α.

Example 2: Consider assertion π from Example 1, we want to turn it to TLM assertion α which is time insensitive an it is formulated over interface variables. From CDFG traversal we know that signal $read0$ (shown in Fig. 9-2) is asserted when F_0 is not empty. Thus, $X(Xread0 = 1) \equiv (\neg F_0.empty)$. Having non-empty F_0 implies that *write* signal of F_0 has been asserted before. Thus, we can rewrite the formula as $XX(F0.read0 = 1) \equiv X(\neg F_0.empty) \equiv (F_0.write)$ Using the knowledge $(F1.data\_in[2..1] = 1 \wedge F_1.write1) \equiv (X(F1.read1) \wedge XX(F1.data\_out1[2..1] = 1)$. we get the following formula:

$\theta : write \rightarrow \mathbf{F}XX(read1 \rightarrow \mathbf{F}Xdata\_out1[2..1] = 1)$

Finally, assertion α (after name mapping) can be written as:

$\alpha : Fwrite \rightarrow \mathbf{F}(read1 \rightarrow \mathbf{F}pkt\_out1.addr = 1)$

9.1.3 Test Generation at TLM Level

An assertion α represents a functional property which holds in the design and violation of it exhibits a design fault. Assertion based test generation methods take property $\neg\alpha$ and use model checkers to generate a counter-example for $\neg\alpha$. In other words, checking property $\neg\alpha$ leads to generate a test which can activate the scenario of the property α. Therefore, proper set of assertions results in higher fault coverage and guarantees the success of property based test generation.

In this research, we make use of SMV as a formal specification to model TLM. SMV model checker [28] is utilized to find the counter-example of property $\neg\alpha$ over SMV model of TLM. The counterexample's assignments to primary inputs is the TLM test case.

9.1.4 Translate TLM tests to RTL tests

The final step is to map TLM test vectors to the RTL tests. Since TLM test lacks the timing information in RTL implementation, they cannot be applied to RTL directly. The mapping process consists of two parts. First, the input/output variables are mapped. Next, templates are utilized to map TLM transactions to sequence of RTL computations. The template enables addition of timing relationship. This process is the inverse of our RTL to TLM assertion mapping. The timing relationship in templates can be provided by the designers or can be extracted by design analysis tools [31].

9.2 Case Studies

We discuss the application of our approach on two case studies: a NoC switch protocol and a pipelined processor. In these experiments we make use of Bounded SMV Model Checker [28] to optimize test generation time.

9.2.1 Wormhole Protocol on NOC Switches

Switches are used as the building block of a Network on Chip (NoC). They receive packets as input and forward it to respective output ports. In this case study, the router uses wormhole routing protocol. We consider test generation for five intersting properties: property 1 is related to reservation of output port of channels, property 2 is about making two internal FIFO's full at the same time, property 3 is about receiving a packet with a specific value, property 4 is related to forcing the acknowledgment signal true and property 5 is related to deadlock detection.

Table 9-1 shows the effectiveness of our method in generating observability-aware tests for these five properties. Since there are no prior efforts for observability-aware post-silicon test generation, we have tried to show the usefulness of our approach in two ways: (i) our approach (with observability constraints) takes reasonable test generation

Table 9-1. Test generation's time for safety properties in network with four switches

Prop.	**Random TG**	**Our Proposed Method**	
		Directed TG	TG with Obs. Constraints
	(min)	(min)	(min)
Property 1	> 600	4.83	6.11
Property 2	> 600	3.45	7.72
Property 3	205	0.49	2.45
Property 4	502.6	1.85	4.73
Property 5	> 600	8.54	13.49

time compared to directed test generation (without observability), (ii) random test generation may be infeasible to activate buggy scenarios and propagate their effects to trace signals. We also tried to generate the test directly from RTL when the RTL is buggy; however, test generation failed because the counterexample cannot be produced. This observation emphasizes the importance of test generation in golden TLM. Table 9-1 provides statistics of test generation on four other selected properties. The column *DirectedTG* shows the needed time for generating directed test without considering observability constraints. The time consumption is comparable with the proposed approach but the effect is not observable on trace signals so these test are not useful for post-silicon. Table 9-1 also shows that TLM random test generation is drastically worse than our approach and in most of the cases it cannot activate the scenario.

9.2.2 Pipelined Processor

We have applied our method on a MIPS processor with 5 stages: Fetch (it fetches the new instruction from memory), Decode (it decodes the instruction and reads the possible operands), Execute (it executes the instruction), MEM (it is responsible for load and store operations) and WriteBack (stores the results in instruction's target register). These stages are implemented by one or more IP block in both RTL and golden TLM implementations. These IP blocks are connected by FIFOs together. The result of test generation based on properties related to testing fetch, decode execution units of the processor are reported in Table 9-2. It is obvious that test generation with observability constraints is most beneficial for post-silicon validation.

Table 9-2. Test Generation's Time for safety properties in a pipelined processor

Prop.	**Random TG**	**Our Proposed Method**	
		Directed TG	TG with Obs. Constraints
	(min)	(min)	(min)
Property 1	> 600	0.83	1.90
Property 2	92.10	1.12	1.17
Property 3	297.05	3.00	7.47
Property 4	416.02	1.12	1.36

9.3 Summary

This chapter presents a high level directed test generation method based on a golden TLM model of the design. The proposed method generates directed test cases at TLM level and maps them back to RTL level to measure the effectiveness of the generated test cases. We generate test cases in a way that not only activate buggy scenarios (especially the scenarios that are hard to activate), but also they guide the effect of buggy scenario to observable points in order to help the debugger to root-cause the source of faults. Our approach has several merits. First, it enables test generation for buggy RTL designs since our tests are generated using golden TLM model. Secondly, our proposed method overcomes the limit of applying direct test generation methods at RTL level since TLM models are significantly less complex than RTL implementation. Thus, the complexity of applying model checkers at TLM level is lower. Finally, our test generation takes observability into consideration by forcing results of the buggy scenario activation to the trace signals. Our case studies using NOC router, Flash protocol and processor designs demonstrated the effectiveness and feasibility of observability-aware test generation.

CHAPTER 10
COVERAGE OF SECURITY PROPERTIES IN RUN-TIME

Post-silicon validation is a major challenge due to the combined effects of debug complexity and observability constraints. Assertions as well as a wide variety of checkers are used in pre-silicon stage to monitor certain functional scenarios. Pre-silicon checkers can be synthesized to coverage monitors in order to capture the coverage of certain events and improve the observability during post-silicon debug. Synthesizing thousands of coverage monitors can introduce unacceptable area and energy overhead. On the other hand, absence of coverage monitors would negatively impact post-silicon coverage analysis. In this chapter, we propose a framework for cost-effective post-silicon coverage analysis by identifying hard-to-detect events coupled with trace-based coverage analysis.

Post-silicon validation techniques consider many important aspects such as effective use of hardware verification techniques [101] and stimuli generation [4]. Several approaches are also focused on test generation techniques [32, 48] that can address various challenges associated with post-silicon debug. There are also some techniques that focus on coverage analysis of some functional events in the manufactured design. They use extra components (coverage monitors) corresponding to assertions in post-silicon validation to address controllability and observability issues in post-silicon as well as to provide a coverage measurement during run-time [8]. The pre-silicon assertions are converted to automatons and gate-level hardware to monitor certain events during post-silicon validation [8, 23]. Coverage monitors can be reconfigured during run-time to change the focus of the observability. Unfortunately, synthesized coverage monitors can introduce unacceptable hardware overhead. Adir et at. proposed a method to utilize post-silicon exerciser on a pre-silicon acceleration platform in order to collect coverage information from pre-silicon [6]. However, the collected pre-silicon coverage may not accurately reflect post-silicon coverage in many scenarios.

Currently there is no effective way to collect coverage of certain events directly and independently on silicon. Engineers need to assume that they will cover at least the same set of post-silicon coverage events as they cover with pre-silicon exercisers using accelerators/emulators [5]. However, we cannot be sure about the accuracy of these coverage metrics since silicon behaves differently than the simulated/emulated design mainly because of asynchronous interfaces. Moreover, because of time constraints, validation engineers are not able to hit all of the desired coverage events during pre-silicon validation or some coverage events are not activated enough. Therefore, they are seeking for an accurate and efficient way to know the coverage of desired events on silicon.

Knowledge of internal signal states during post-silicon execution helps to trace the failure propagation to debug the circuit. Trace buffers are used to sample a small set of internal signals since they can help in restoration of other signals and improve the design observability. There are different techniques to select trace signals such as structure/metric-based selection [12], simulation-based selection, as well as hybrid of both approaches [87]. Recently, Ma et al. have proposed a metric that models behavioral coverage [95]. However, none of these approaches consider functional coverage analysis as a constraint for signal selection. Our proposed signal selection improves functional coverage analysis without compromising the debugging observability.

This chapter makes three major contributions. We propose a method to utilize existing debug infrastructure to enable coverage analysis in the absence of synthesized coverage monitors. This analysis enables us to identify a small percentage of coverage monitors that need to be synthesized in order to provide a trade-off between observability and design overhead. To improve the observability further, we also present an observability-aware trace signal selection algorithm that gives priority to signals associated with important coverage monitors. Our experimental results demonstrate that an effective combination of coverage monitor selection and trace analysis can maintain the debugging observability with drastic reduction (up to 10 times) in the required coverage monitors.

Although our proposed method can provide coverage data only for the recorded cycles (instead of the complete execution), it can significantly reduce the post-silicon validation effort for various reasons. First, the traced data for specific cycles are able to restore untraced signals for additional cycles. Moreover, when we know that certain events have been hit (covered) by trace analysis, the validation effort can be focused on the remaining set of events. From a practical perspective, it is not valuable to collect coverage data when generating testcase for an exerciser [80] or during checking a testcase since the exerciser code is relatively simple, repetitive and not expected to hit bugs on silicon. If the trace buffer is reconfigured to record signals during testcase execution, more insight about coverage events can be obtained.

The remainder of the chapter is organized as follows. Section 10.1 provides an overview about assertion-based validation. Section 10.2 describes our post-silicon functional coverage analysis framework. Section 10.3 presents our experimental results. Finally, Section 10.4 concludes the chapter.

10.1 Overview on Assertions

Based on the functional coverage goal, a design is instrumented to check specific conditions of few internal signals. For example, assertions are inserted in a design to monitor any deviation from the specification. These days, designers mostly use one of the powerful assertion languages such as PSL (Property Specification Language) to describe interesting behavioral events (linear temporal logic assertions). First, we give an overview of PSL assertions and then we describe our method using them. However, the presented method in this chapter is not dependent on any assertion language. There are two general types of assertions: *assert* and *cover* statements. There is a single bit associated with *assert* which indicates the pass or fail status of the assertion. The *cover* assertion triggers at the end of the execution when the assertion is not covered during the run-time. PSL assertions contain several layers such as Boolean and temporal layers and it can be used on top of different HDL languages including Verilog and VHDL. It denotes temporal sequence using different operators such as ";" (notation of one clock cycle step), ":" operator for concatenation and operators $[low : high]$, $[*]$ or $[+]$ present the notation of bounded or unbounded repetition. Operator $[*]$ shows the repetition of zero or more instances; however, $[+]$ denotes one or more repetition. Operators "&" and "|" show the logical AND and OR between sequences. Different operators such as *always*, *eventually*! and implication and non-overlapped implication which means right hand side property can hold one cycle after occurrence of left hand side sequence can be combined with other arguments and operators. Assertions are usually accompanied by an active edge of the clock (usually the rising edge of the clock is considered as default). PSL assertion can include Boolean expressions (b_i), sequences of events of Boolean primitives ($s_i = b_l; ...; b_r$) and properties on Boolean expressions and sequences.

Assertions can be classified into two groups: conditional and obligation [24]. The goal of a conditional assertions is to detect a failure. Therefore, it is activated every time all of its events are observed. For example, "$assert \quad never \quad b_1$" is called a conditional assertion as every time b_1 is evaluated true, a failure will happen and assertion should be triggered. On the other hand, an assertion is in obligation mode when a failure in its sequence triggers it. For example, "$assert \quad always \quad b_2$" is in obligation mode as b_2 should be always true and every time it is evaluated to false, the assertion is activated.

Example 1: Consider a part of a circuit shown in Figure 10-1. Suppose that we have two design properties: first, whenever signal E asserted, signal H is supposed to be asserted within next three cycles. The following assertion describes this property A_1 : $assert \quad always(E \rightarrow \{[*1:3]; H\}) \quad @rising\_edge(clk)$. Consider a second property where we would like to cover functional scenarios such that D and I signals are not true at the same time. This property can be formulated as $A_2 : assert \quad never(D\&I)$.

During post-silicon, assertions are synthesized to checker circuits (coverage monitors) which are responsible to check specific properties during run time. In other words, checkers are extra components corresponding to assertions that are used in post-silicon

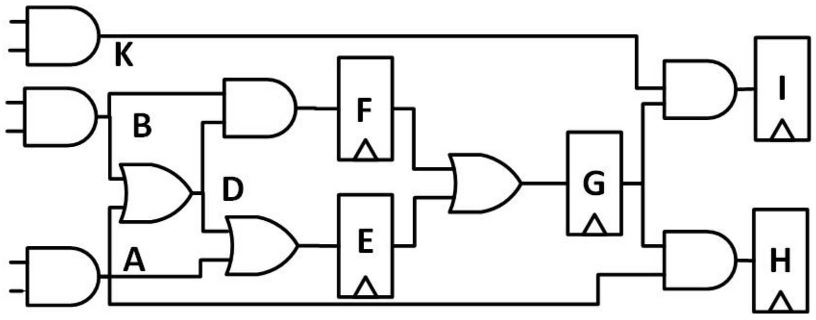


Figure 10-1. A simple circuit to illustrate design properties

validation to increase the observability as well as provide a coverage measurement technique.

10.2 Post-Silicon Functional Coverage Analysis

To have full observability in post-silicon, one option is to synthesize all of the functional scenarios (typically thousands of assertions, coverage events, etc.) to coverage monitors and track their status during post-silicon execution. However, this option is not practical due to unacceptable design overhead. Therefore, designers would like to remove all or some of the coverage monitors to meet area and energy budgets. It creates a fundamental challenge to decide which coverage monitors can be removed. We propose an approach to evaluate the assertion activation efforts by on-chip trace buffer and rank them based on the difficulty in covering/detecting them. Clearly, the hard-to-detect ones should be synthesized, whereas the easy-to-detect ones can be ignored (trace analysis can cover them). Figure 10-2 shows an overview of our proposed method. Our proposed approach consists of four major steps: decomposition of coverage scenarios, signal restoration, coverage analysis and signal selection. The remainder of this section describes these steps.

10.2.1 Decomposition of Coverage Scenarios

Suppose that a gate-level design D as well as a set of pre-silicon RTL assertions $\mathbb{A}$ are given and our plan is to use trace buffer information to determine activation of $\mathbb{A}$ in model D during silicon execution. We propose an off-line functional event decomposition to enable post-silicon coverage analysis. The decomposition can be done in pre-silicon. First,

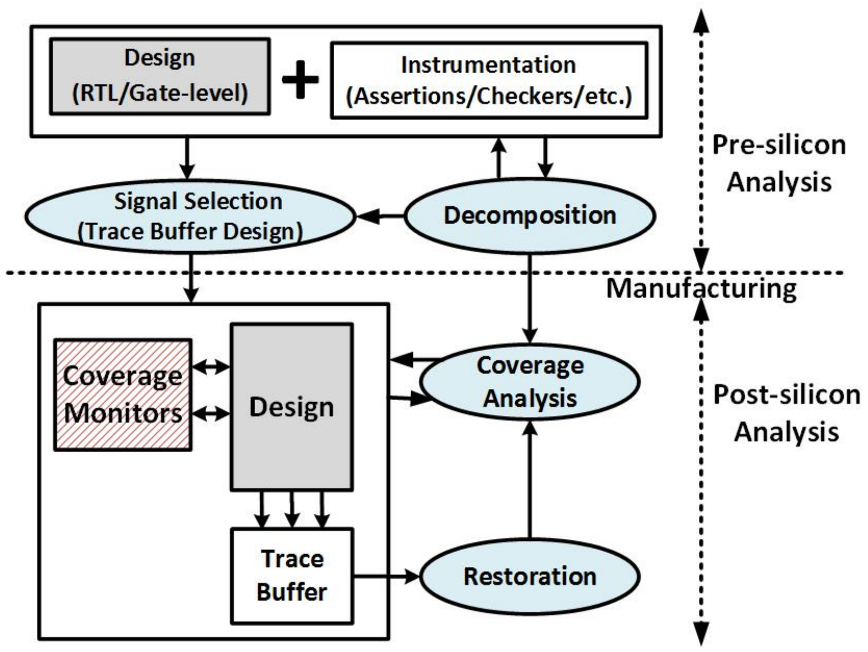


Figure 10-2. Overview of our proposed approach.

RTL assertions are scanned to extract their signals and their corresponding gate-level signals based on name mapping methods. Next, each RTL assertion from set $\mathbb{A}$ is mapped to a set of clauses such that each clause contains assignments to a set of signals in specific cycles.

Formally, each pre-silicon assertion $A_i \in \mathbb{A}$ is scanned and its signals and its corresponding gate-level signals are defined. Then, A_i is decomposed to a set of clauses $A \equiv \mathbb{C} = \{C_1, C_2, ..., C_n\}$ based on its mode (conditional or obligation). Each C_j can be formalized as $C_j = \{\alpha_1 \Delta_1 \alpha_2 \Delta_2 ... \Delta_{m-1} \alpha_m\}$. Each α_k presents a Boolean assignment on gate-level signal $n \subset \mathbb{N}$ ($\mathbb{N}$ shows all corresponding gate-level signals of set $\mathbb{A}$) on cycle c_t where $1 \leq c_t \leq CC$ (Suppose we know that the manufactured design will be simulated for maximum CC clock cycles) such as $\alpha_k : \{n = val \quad in \quad cycle[c_t]\}$ where $val \in \{0, 1\}$. Operators Δ_k can be one of the logical operations such as AND, OR or NOT. As a result, the original assertion is translated as a set of clauses $\mathbb{C}$ in a way that activation of one of them triggers the original assertion.

From now on, we assume that signals of A_i are mapped to corresponding assertion on gate-level signals. In order to generate each of C_j, Algorithm 17 is used. It partitions assertions based on their mode. If an assertion is in obligation mode, its original conditions are negated since we are looking for conditions that cause the assertion to fail (lines 7-8) and each C_j contains a subset of the negated conditions over different clock cycles that cause the assertions to fail. On the other hand, if the assertion is in conditional mode, the original assertions are kept as they are (lines 9-11). Each clause contains a subset of conditions which is logically equivalent to a set of conditions (lines 12-13). Clauses are also expanded over time and contain timing information (line 14). Therefore, each assertion is mapped to a set of clauses such that activation of one of the clauses leads to activation of the original RTL assertion (line 15). Specifically, the following rules are used to generate the set of clauses ($\mathbb{C}$):

- If an assertion is in obligation mode and it contains an AND operator ($p \land q$), the operands are negated and AND will be changed to a OR operator ($p \lor \ q$). For example, in assertion "*assert always p & q*", whenever conditions p or q are false, the assertion is activated. Therefore, the assertion is translated to a set of clauses as $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 0[t] \lor q = 0[t]\}$ (clauses are also expanded over time).
- If an assertion is in obligation and it contains OR operator such as:

  $$assert \quad always \quad (p \quad | \quad q)$$

  The conditions will be negated and the set of clauses is extracted as: $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 0[t] \land q = 0[t]\}$.
- If an assertion is in obligation mode and it contains an implication operator, antecedent conditions are not modified. However, consequent conditions are negated. For example, if we have "*assert always* $(p \rightarrow$ *next* $q)$", it is converted to $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 1[t] \land q = 0[t+1]\}$. Next operation shows its effect in condition $q = 0[t+1]$.

- If an assertion is in conditional and it contains OR operator such as

  $$assert \quad never \quad (p \quad | \quad q)$$

  It is translated to $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 1[t] \vee q = 1[t]\}$.

- If an assertion is in conditional mode and it contains an AND operator $(p \wedge q)$, the operation is kept as it is. For example, in assertion "$assert \quad never \quad (p \quad \& \quad q)$", if p and q are true at the same time, the assertion will be activated. Therefore, the assertion is translated as $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 1[t] \wedge q = 1[t]\}$.

- If an assertion is in conditional mode and it contains implication operator, antecedent and consequent conditions remain the same as the original assertion. For example, if we have "$assert \quad never \quad (p \rightarrow next \quad q)$", it is converted to $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 1[t] \wedge q = 1[t+1]\}$.

- If there are *eventually!* or *until* operators in an assertion, based on the mode of assertion it shows its effect in generating repeating conditions in different clock cycles. For example, "$assert \quad always \quad (p \rightarrow \quad eventually \quad q)$" is translated to $\mathbb{C} = \bigcup_{t=1}^{CC}\{(p = 1[t]) \wedge (q = 0[t] \wedge q = 0[t+1] \wedge ... \wedge q = 0[CC])\}$. On the other hand, if we have an assertion as "$assert \quad always \quad (p \quad until \quad q)$", the conditions are found as: $\mathbb{C} = \bigcup_{t=1}^{t+n=CC}\{(p = 1[t]) \wedge (p = 0[t+1] \vee p = 0[t+2] \vee ... \vee p = 0[t+n-1]) \wedge (q = 1[t+n])\}$.

Example 2: Consider the assertions in Example 1 form Section 10.1. We assume that the circuit will be executed for 10 clock cycles for post-silicon validation. The first property (A_1) will be decomposed to equivalent conditions as follows:

$\mathbb{C}_{A_1} : \{\{E = 1[1] \wedge H = 0[2] \wedge H = 0[3] \wedge H = 0[4]\}, \{E = 1[2] \wedge H = 0[3] \wedge H = 0[4] \wedge H = 0[5]\}, ..., \{E = 1[7] \wedge H = 0[8] \wedge H = 0[9] \wedge H = 0[10]\}\}$

The assertion is activated if signal E is asserted and signal H remains false for next three cycles. The second property is decomposed as shown below since it will be activated if both D and I are true at the same time.

$\mathbb{C}_{A_2} : \bigcup_{t=1}^{10}\{D = 1[t] \wedge I = 1[t]\}$

Algorithm 17: Assertion decomposition algorithm

```
1: **Input:** RTL assertions $\mathbb{A}$
2: **Output:** $\mathscr{C}$ which maps each $A_i \in \mathbb{A}$ to equivalent $\mathbb{C}$
3: $\mathscr{C} = \{\}$
4: for each $A_i \in \mathbb{A}$ do
5:   $\mathbb{C} = \{\}$
6:   if $A_i$ is in obligation mode then
7:     $\Omega$ = FindFailureConditions($A_i$, $\mathbb{G}$)
8:   else
9:     /*$A_i$ is in conditional mode*/
10:    $\Omega$ = FindPassingConditions($A_i$, $\mathbb{G}$)
11:  end if
12:  for every possible case do
13:    $\mathbb{C} = \mathbb{C} \cup$ SubsetOfEquivalentConditions($\Omega$)
14:    addTiming($\mathbb{C}$)
15:  end for
16:  $\mathscr{C}.put(A_i, \mathbb{C})$
17: end for
18: Return: $\mathscr{C}$
```

The computed conditions are used to detect activation of assertions during post-silicon validation as described in Section 10.2.3.

10.2.2 Restoration of Signal States

Trace buffers can be efficiently used to sample small number of signals during certain number of clock cycles. The stored data can be used to restore other internal signals' value and the design states. The design states are used to detect functional events' activation. Suppose that we have a gate-level design with $\mathbb{G}$ internal signals and the design has been executed for CC clock cycles during post-silicon validation. A set of signals ($\mathbb{S} \quad where \quad \mathbb{S} \subset \mathbb{G}$) are sampled and their values are stored in trace buffer T during post-silicon execution for CC_t clock cycles ($CC_t \leq CC$). The information of the trace buffer (with $|\mathbb{S}|$ and CC_t dimensions) can be used to find the values of other signals ($\mathbb{G} - \mathbb{S}$). The restoration starts from the stored values of $\mathbb{S}$ signals over CC_t cycles and go forward and backward to fill the values of matrix $\mathbb{M}_{\mathbb{G}xCC}$. Matrix $\mathbb{M}$ is used to present states of the design during CC clock cycles. Each cell of matrix $\mathbb{M}$ can have value 0, 1 or

X. Value X in $m_{i,j} \in \mathbb{M}$ presents the fact that the value of signal i in clock cycle j cannot be restored based on traced values of $\mathbb{S}$ sampled signals. We utilize matrix $\mathbb{M}$ information to determine if any of assertions is definitely covered during run-time.

We consider all signals while computing restoration ratio. Instead of counting the number of flip-flops that have valid value over different cycles [12], we count each individual internal signal that has either 0 or 1 value in a specific clock cycle (each cell of matrix $\mathbb{M}$ that has a non-X is counted) and divide it over the total number of internal signals $\mathbb{G}$ multiplied by CC.

10.2.3 Coverage Analysis

Our plan is to use both traced and restored values to check the clauses that we found in Section 10.2.1 to define easy-to-detect assertions. In order to find coverage for assertions in set $\mathbb{A}$, we consider each assertion $A_i \in \mathbb{A}$ and find its corresponding decomposed clauses, set $\mathbb{C}$, as described in Section 10.2.1. Set $\mathbb{C}$ is designed in a way that if one of the $C_i \in \mathbb{C}$ can be evaluated to true on matrix $\mathbb{M}$, assertion A_i is triggered. Using the proposed method, each C_i contains a set of Boolean functions (α_j) and each $\alpha_j : n = val$ in cycle t where $1 \leq t \leq CC$ is mapped to one cell of matrix $\mathbb{M}$ ($m_{n,t}$). If the value of $m_{n,t}$ is equal to $val \in \{0, 1\}$, the condition α_j is evaluated true. Condition C_i is evaluated true when the expression consisting of all α_j and Δs evaluated to be true. An assertion is called covered during post-silicon validation if one of its C_is is evaluated to be true.

For assertions that originally contains implication operator ($A : assert \quad p \rightarrow q$), we keep the information that which Boolean α_j belongs to the precondition (p) and which conditions belongs to the fulfilling condition (q) when we want to check their conditions over $\mathbb{M}$. In order to check assertion A, we start from rows which belongs to signals existing in antecedent and check every cycle to find the desired value. Then, we continue the search for consequent from those cycles when antecedent is true to find values that makes whole A true. In other words, to be able to find out the activation of assertion A, we need to minimize the number of X values in cells of matrix $\mathbb{M}$. We also count how many times assertion A is activated for sure. Note that for checking conditions, 3-valued (ternary) logic is used. In other words, condition $p \vee q$ is evaluated as true if signal p is true and q has X value and vice versa. Algorithm 18 presents the coverage analysis procedure. It counts the number of times an assertion is activated during execution. If *activationCount* is equal to zero, it means that we cannot determine activation of the assertion based on trace buffer values. The coverage (*cov*) percentage is computed by counting the assertions that have been activated at least once and dividing it by the number of total assertions.

Algorithm 18: Assertion coverage measurement algorithm

```
1: Input: Trace buffer $T$, trace signals $\mathbb{S}$, gate-level signals $\mathbb{G}$, condition map $\mathscr{C}$,
   assertions $\mathbb{A}$, max cycle $CC$
2: Output: Coverage map $\Theta$
3: $T$=Restoration($\mathbb{S}$);
4: $\mathbb{M}$ = ConstructDesignStatesMatrix($T, \mathbb{G}, CC$)
5: for each $A_i \in \mathbb{A}$ do
6:    activationCount = 0
7:    $\mathbb{C}$ =$\mathscr{C}$.get($A_i$)
8:    for each $C_j \in \mathbb{C}$ do
9:       if checkCondition($C_j, \mathbb{M}$) then
10:         activationCount++
11:      end if
12:   end for
13:   $\Theta$.put($A_i$, activationCount)
14: end for
15: Return: $\Theta$
```

Example 3: Consider the circuit shown in Figure 10-1 and the associated assertions shown in Example 1. Suppose that the only two signals (A and B) can be traced during post-silicon validation (the width of trace buffer is two). Note that in signal selection part we are not limited to flip-flops and every internal signal can be considered as potential sampled signal. Table 10-1 shows the states of the design based on the stored values of A and B signals. In fact, Table 10-1 shows matrix $\mathbb{M}$. The restoration ratio is equal to: $67/(9 * 10) = 0.74$. In other words, 74.44% of internal states are restored. Suppose that we take the clauses shown in Example 2 and the matrix shown in Table 10-1 as inputs to compute the coverage of these assertions during run-time. Based on information shown

Table 10-1. Restored signals for circuit shown in Figure 10-1 when A and B are trace signals.

Signal/Cycle	1	2	3	4	5	6	7	8	9	10
A	0	1	0	1	1	0	0	0	1	0
B	1	0	0	1	0	0	0	1	1	1
K	X	X	X	X	X	X	X	X	X	X
D	1	1	0	1	1	0	0	1	1	1
E	X	1	1	0	1	1	0	0	1	1
F	X	1	0	0	1	0	0	0	1	1
G	X	X	1	1	0	1	1	0	0	1
I	X	X	X	X	X	0	X	X	0	0
H	X	0	X	0	1	0	0	0	0	0

in Table 10-1, assertion A_1 is activated since signal E is asserted in cycle 6 and signal H remains zero in the next three cycles, 7, 8 and 9 (activation of A_1 is detected twice). However, we cannot comment on A_2 since the respective conditions cannot be evaluated.

Until now, we identified which assertions are activated during run-time for sure. We rank assertions based on the required efforts to detect them using our proposed method of Section 10.2.3 to decide which assertions are better to be kept as coverage monitors in post-silicon to improve the design observability and increase the assertion coverage. The assertions that are hard-to-detect (for example, cannot be detected even one time using the proposed method) or represent critical functional scenarios are best candidates to be kept as coverage monitors in silicon to improve the design observability. Algorithm 19 presents the proposed approach. The algorithm sorts set $\mathbb{A}$ based on their activation count that obtained from Algorithm 10.2.3 and priority (critical scenarios or assertions that their activation cannot be detected using trace buffer values have higher priority). Next, we select assertions that fits in area and power budgets and increase total coverage and add them to cov_mon. For example, if two assertions have same priority, we choose the one that has less number of operators and signals (represents less area overhead). The algorithm returns the set cov_mon as selected coverage monitors.

Algorithm 19: Coverage monitor selection algorithm

```
1: **Input:** Assertions $\mathbb{A}$, desired coverage $des\_cov$, budget for coverage monitors $budg$,
   gate-level signals $\mathbb{G}$, trace buffer $T$ with trace signal $\mathbb{S}$, trace buffer width $W$
2: **Output:** Selected coverage monitors $cov\_mon$
3: $cov\_mon = \{\}$
4: $\mathscr{C}$=DecomposeAssertions($\mathbb{A}$)
5: $\Theta$=CoverageAnalysis($T, \mathbb{S}, \mathbb{G}, \mathscr{C}, \mathbb{A}, CC$)
6: $cov$=findCoverageNumber($\Theta$,$\mathbb{A}$)
7: $\mathbb{U}$=findUndetectedAssertions($\Theta$,$\mathbb{A}$)
8: SortBasedOnDetectionEfforts($\mathbb{A}$,$\Theta$)
9: $tmp\_cost = cost$, $tmp\_cov = cov$
10: **while** $cov\_t < des\_cov$ && $cost\_t < budg$ **do**
11:   find $a_i \in \mathbb{A}$ where $a_i.selected = false$ and $budg - cost\_t - a_i.cost \geq 0$
12:   $cost\_t += a_i.cost$
13:   $a_i.selected = true$
14:   **if** $a_i \in \mathbb{U}$ **then**
15:     $cov\_t++$
16:   **end if**
17:   $cov\_mon = cov\_mon \cup u_i$
18: **end while**
19: **Return:** $cov\_mon$
```

10.2.4 Coverage-aware Signal Selection

Traditional signal selection methods select signals that have priority over other design signals as they may have a better restorability and more internal signals might be restored during the off-chip analysis. However, if we select trace signals that have better restorability on signals appear on assertions, we can increase the chance of finding the activation of assertions. We propose a signal selection algorithm emphasizes restorability of assertion signals to be able to improve the assertion-based coverage analysis. We show that this way of trace signal selection has a better impact of analysis of assertion coverage while it has a negligible effect on observability of the whole design.

Algorithm 20 shows our proposed signal selection algorithms that improves assertion coverage analysis. In order to select trace signals that have a better restorability on assertion signals, pre-silicon assertions $\mathbb{A}$ are scanned to find their signals (set $\mathbb{N}$) and their importance based on how many times a specific signal is repeated in map Ψ (lines 4 and

Algorithm 20: Assertion-aware trace signal selection algorithm

```
1:  **Input:** Assertions $\mathbb{A}$, trace buffer width $W$, gate-level signals $\mathbb{G}$
2:  **Output:** Selected Trace Signals $\mathbb{S}$
3:  $\mathbb{N}$=findSignalsExistInAssertions($\mathbb{A}$)
4:  $\Psi$ = findNumberOfOccuranceOfEachSignal($\mathbb{N}$)
5:  $\mathbb{S} = \{\}$
6:  while $\mathbb{S}.size() < W$ do
7:     generate random tests $I$
8:     for each $g_i \subset \mathbb{G}$ which is not in $\mathbb{S}$ do
9:        calculate restoration of $g_i \cup \mathbb{S}$
10:    end for
11:    select $n_i$ with maximum restorability on $\mathbb{N}$.
12:    $\mathbb{S} = \mathbb{S} \cup n_i$
13: end while
14: Return: $\mathbb{S}$
```

5). We modify existing simulation-based trace signal selection algorithm to select signals which has maximum restoration ratio on assertion signals ($\mathbb{N}$) based on the simulated values of random test vectors for several cycles. If there is a tie, we use the signal that has a higher value in set Ψ. The algorithm continues until it selects as many as W trace signals (lines 6-12). Please note that our approach (providing emphasis on assertion signals) can be applied on top of other signal selection algorithms as well.

Example 4: As it can be seen from Example 3, the activation status of assertion A_2 cannot be detected based on information of Table 10-1. Using Algorithm 3, only A and I are selected as trace signals based on their good restorability on assertion signals $(E, H, D \quad and \quad I)$. Restoration and coverage analysis (using the traced values of new signals) would be able to detect activation of both assertions in Example 1.

10.3 Experiments

10.3.1 Experimental Setup

In order to evaluate the efficiency of our proposed approach, we have implemented our assertion decomposition, restoration, coverage analysis and signal selection algorithms using C++. We have applied our proposed methods on ISCAS'89 benchmarks (since most of the existing signal selection algorithms work with only these benchmarks). The trace buffers were chosen with a widths of 8, 16 and 32 and depth of 1024[1] . We have generated assertions both in obligation and conditional modes based on the presented method of [88]. Assertions were decomposed as a set of clauses using Algorithm 17. We simulated the benchmarks using different trace buffers for 1024 clock cycles with random test vector to model post-silicon validation. Trace signals were chosen based on our implementation of the presented method in [15] since it is the most recent signal selection approach. In the next step, we dumped the stored values of trace buffer and we tried to restore the values of unsampled signals over different clock cycles to construct the matrix representing the states of the circuit. Next, the assertion conditions were checked over matrix and we counted the number of activated assertions during run-time based on Algorithm 18. Finally, we used the signals selected by Algorithm 3 to further improve functional coverage analysis. Moreover, we use Algorithm 19 to selectively synthesize some coverage monitors which are more beneficial for improving functional coverage.

10.3.2 Results

Table 10-2 presents results for assertion coverage of total 12000 (4000 for each trace buffer configuration) assertions for each benchmark. The first three columns show the type of the benchmark, the number of its gates and the width of its trace buffer, respectively. The fourth column shows the restoration ratio based on existing trace signal selection. The fifth column shows the coverage of assertions using trace buffer (without introducing any overhead). Note that, *Observability-aware SS* represents our assertion coverage analysis framework on top of existing signal selection techniques. We improved the assertion coverage using our proposed signal selection algorithm with negligible effect on restorability of whole design (sixth and seventh columns). Note that, *Coverage-aware SS* represents our assertion coverage analysis framework on top of our coverage-aware signal

[1] A trace buffer with width 32 and depth 1024 represents that it can trace the values of 32 signals over 1024 clock cycles.

selection method. Since signal selection algorithms are based on heuristic methods, in some of the cases, our coverage-aware signal selection algorithms improves the restorability of the design (such as s15850).

If we zoom in on each row of Table 10-2, the activation details of each type of assertions are reported in Table 10-3. The first three columns of Table III show the type of the benchmark, the number of its gates and the width of its trace buffer, respectively. The fourth and ninth columns show the restoration of the design using existing trace signal selection method as well as our proposed signal selection algorithms, respectively. The fifth and tenth columns show the coverage of one thousand single variable assertions (consisting of only one condition) on specific clock cycles. The sixth and eleventh columns show the coverage of one thousand two-variable assertions with AND operators where their conditions have templates of $\{n_1 = val_i \quad [c_i] \& \quad n_2 = val_j \quad [c_j]\}$ using both signal selection methods. The seventh and twelfth columns show the coverage of one thousand three-variable assertions with AND operators where their conditions are in the form of $\{n_1 = val_i \quad [c_i] \& \quad n_2 = val_j \quad [c_j] \& \quad n_3 = val_t \quad [c_t]\}$ respectively. The eighth and thirteenth columns show the coverage of one thousand three-variable assertions with OR operators where their conditions are in the form of $\{n_1 = val_i \quad [c_i] || \quad n_2 = val_j \quad [c_j] || \quad n_3 = val_t \quad [c_t]\}$ respectively. The results demonstrated the fact that using our approach enables designers to achieve significant functional coverage (up to 93%, 58% on average) without synthesizing any coverage monitors.

10.3.3 Observability versus Hardware Overhead

The results shown in Tables 10-2 and 10-3 show the extent of functional coverage analysis without introducing any hardware penalty for synthesized coverage monitors. In other words, the reported coverage can be achieved for free during post-silicon debug. Here, we talk about the trade off between the number of assertions we can keep in the post-silicon as coverage monitors and the number assertions we delete. However, if designers have a budget to tolerate extra power and area overheads of few coverage

Table 10-2. Assertion coverage when the total number of assertions for each row is 4000 (12,000 per benchmark)

Benchmark			Signal Selection			
			Observability-aware SS		Coverage-aware SS	
Type	#gates	#Traces	Restoration%	Asser. Cov.%	Restoration%	Asser. Cov.%
S5378	2995	8	60.97	46.97	58.57	49.6
		16	79.27	63.6	76.95	64.23
		32	93.10	88.5	92.26	90.57
S9234	5844	8	84.85	65.4	76.03	65.8
		16	90.19	75.27	83.70	83.12
		32	94.54	90.67	93.8	93.45
s15850	10383	8	72.03	59.7	76.03	65.8
		16	80.97	61.6	75.55	68.7
		32	84.14	72.9	82.66	74.9
s35932	17828	8	41.09	26.825	41.63	27.02
		16	41.35	26.825	41.88	27.05
		32	41.79	26.825	42.22	27.25
s38417	23843	8	36.53	23.025	36.97	23.23
		16	43.76	28.575	46.91	32.58
		32	49.77	35.075	55.82	42.33
s38584	20717	8	72.97	47.625	67.78	59.8
		16	79.15	63.65	76.53	69.3
		32	88.85	73.65	87.27	82.78

Table 10-3. Assertion coverage using trace buffer information

			Observability-aware SS					Coverage-aware SS				
Benchmark	#gates	#Trace	% restored Gates	% Single Variable Assertion	% Two Variable Assertion (AND)	% Three Variable Assertion (AND)	% Three Variable Assertion (OR)	% restored Gates	% Single Variable Assertion	% Two Variable Assertion (AND)	% Three Variable Assertion (AND)	% Three Variable Assertion (OR)
S5378	2995	8	60.97	63.6	36.2	25.4	62.7	58.57	64.2	42.6	25.3	66.3
		16	79.27	77.5	56.3	41.9	78.7	76.95	79.6	58.8	44	74.5
		32	93.10	92.6	86.5	80.9	94.0	92.26	94.5	88.4	84.8	94.6
s9234	5844	8	84.85	77.9	59.7	45	79	76.03	88.0	77.4	68.1	87.5
		16	90.19	85.5	71.1	58.9	85.6	83.7	88.6	81.4	72.6	89.9
		32	94.54	93.5	90.3	84.8	94.1	93.8	94.8	90.0	84.5	93.7
s15850	10383	8	72.03	73.9	54.8	37.1	73.0	76.03	76.5	61.7	46.9	78.1
		16	80.97	75.1	55.5	40.1	75.7	75.55	81.1	63.1	50.2	80.7
		32	84.14	82.2	68.1	56.6	84.6	82.66	84.2	70.6	59.5	85.3
s35932	17828	8	41.09	41.3	16.2	7.7	42.1	41.63	40.9	17.3	7.4	42.5
		16	41.35	41.3	16.2	7.6	42.2	41.88	40.9	17.3	7.4	42.5
		32	41.79	41.3	16.2	7.7	42.1	42.22	41.3	17.4	7.5	42.8
s38417	23843	8	36.53	38.2	13.8	3.9	36.2	36.97	37.7	13.9	5.8	35.9
		16	43.76	44.5	19.7	8.1	42	46.91	46.91	24.2	10.5	48.7
		32	49.77	51.2	26.5	12.4	50.2	55.82	56.3	35.5	19.2	58.3
s38584	20717	8	72.97	60.7	41.2	24.1	72.97	67.78	70.8	54.3	37.7	76.4
		16	79.15	75.3	59	43.4	76.9	79.69	79.7	65.3	51.2	81
		32	88.85	83.5	68.5	56.7	85.9	87.27	90.0	79.5	71.7	89.9

monitors, the primary challenge would be to determine the assertions that should be selected and synthesized in hardware. Figure 10-3 shows coverage improvement if we randomly choose 10% to 90% of the remaining assertions that we cannot be sure about their activation using trace buffer information. The straight line shows the coverage when our method is not used and observability is provided only by using synthesized coverage monitors (the percentage of observability is equal to the percentage of synthesized assertions). On the other hand, we used Algorithm 19 to select hard to detect coverage monitors.

As it can be seen in Figure 10-3 and Figure 10-4, 100% observability is achieved with significant reduction in overhead (40-50% coverage monitors with observability-aware signal selection can provide 100% functional coverage). Figure 10-4 shows the result of observability of s9234 with different trace buffer widths and different coverage monitor selection strategies (the straight line is cut on 50% for improved illustration). As it can be seen, when our signal selection algorithm was used to choose 32 trace signals and our coverage monitor algorithm was used, synthesizing only 10% assertion leads to 100% observability. Although, we presented the result for s9234, we obtained similar results for other ISCAS89 benchmarks.

It can be argued that it takes little effort to cover the first 90%, but significantly more to cover the remaining 10%. Based on our proposed method, if the remaining 10% of the assertions are synthesized, 100% coverage is achieved. However, if it is not possible to synthesize those assertions due to design constraints, increasing the width or depth of trace signals can be considered. Dynamic signal selection capability (if available) can be utilized to focus in tracing of the remaining 10% assertions.

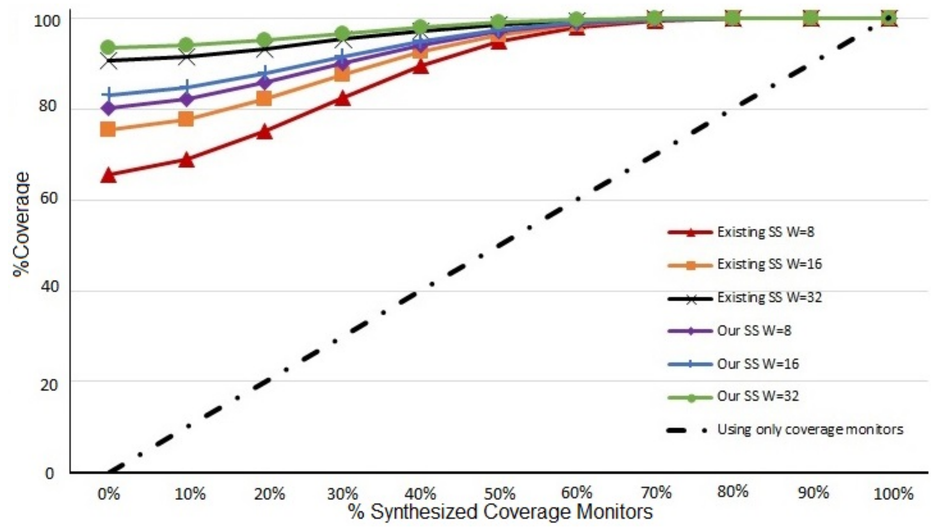


Figure 10-3. Coverage analysis for s9234: coverage monitors are selected randomly.

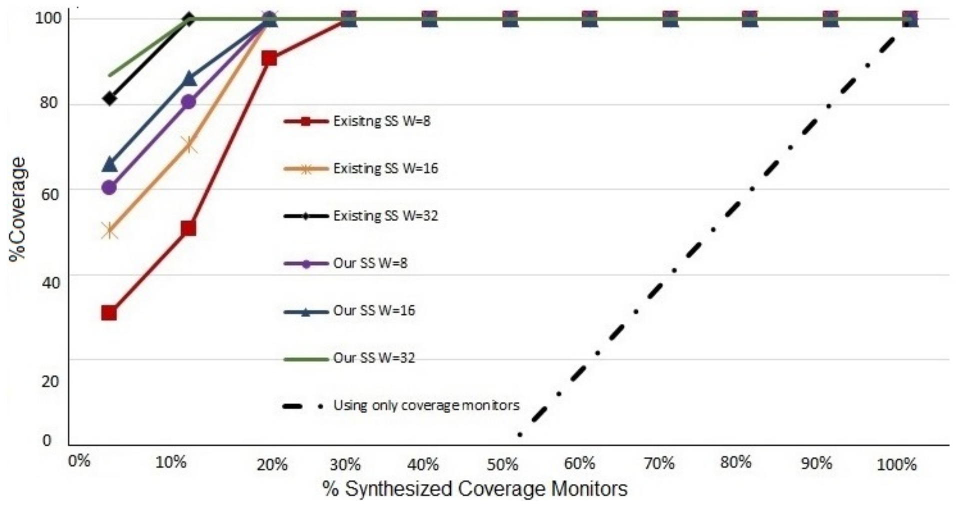


Figure 10-4. Coverage analysis for s9234: coverage monitors are from hard-to-detect events.

The experimental results demonstrated three important aspects of our approach. We provided a technique to improve the design observability when designers have a limited budget for synthesized coverage monitors. We showed that if they synthesize hard-to-detect assertions, the observability improves significantly. Our assertion-aware signal selection algorithm improves the assertion-coverage compared to existing signal selection techniques.

10.4 Summary

We presented an approach to efficiently find functional coverage on silicon without introducing any overhead. The proposed method utilizes the existing debug infrastructure in modern designs to rank coverage monitors in terms of required efforts to detect them. We proposed a framework for trace-based functional coverage analysis. We explored the trade-off between observability and design overhead of synthesized coverage monitors. We also introduced a signal selection algorithm to improve the coverage analysis with negligible impact on restoration ratio. Our experimental results demonstrated that efficient ordering and selection of coverage monitors can drastically reduce (up to 10 times) design overhead without sacrificing functional coverage.

CHAPTER 11
CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

We are living in a connected world where a wide variety of computing and sensing components interact with each other. Safe connectivity is essential in the fabric of Internet-of-Things (IoT) as intelligent computing devices are increasingly embedded in every possible devices in our daily life such as wearable devices (e.g, fitness trackers, smart watches, and glasses), autonomous vehicles and smart homes. These devices are recording, collecting, analyzing, and communicating some of our most intimate personal information such as health-related data in order to provide a real-time aid in daily basis. Unlike microcontroller based designs in the past, even resource constrained IoT devices nowadays incorporate one or more complex System-on-Chips (SoCs). Any failure of security and trust requirements of SoCs may endanger human life and environment by causing damages to critical infrastructure, violating personal privacy, or undermining the credibility of a business.

The risk of cyber-attacks has increased more than anytime before. Attacks on hardware can be more effective and efficient than traditional software attacks since patching is extremely difficult (almost impossible) on hardware designs. Therefore, a security attack can be successfully repeated on every instance of a vulnerable SoC. It is a major challenge to verify the security requirements of SoCs in IoT devices, primarily due to the fact that SoCs are designed using hardware Intellectual Property (IP) blocks to reduce cost while meeting aggressive time-to-market constraints. In this dissertation, I addressed challenges in SoC security validation using effective combination of formal methods as well as test generation approaches.

With the globalization of the IC industry, the outsourcing and integration of third-party hardware Intellectual Property (IPs) has become a common practice for System-on-Chip (SoC) design. However, it raises major security concerns as an attacker can insert malicious components (hardware Trojans) in third-party IPs and tamper the system. Hardware Trojans are inactive most of the time and can be triggered under very rare input sequences. Therefore, using conventional validation method is not effective to detect Trojans. In Chapter 4, I proposed use of Gröbner basis theory to formulate the equivalence checking problem as an ideal membership testing in an algebraic domain and find Trojan-inserted IPs in a deterministic and efficient way as word-level representations are involved. In the proposed approach, equivalence checking formulation starts with converting the design specification as well as its implementation to polynomials, f_{spec} and set F, respectively. Polynomial division is deployed to detect Trojans by reducing the f_{spec} over implementation polynomials (F) until it leads to either a zero remainder or a polynomial that contains only primary inputs. A non-zero remainder indicates that the implementation is not trustworthy. I have applied this technique to large arithmetic IPs with different architectures and the experimental results show that the proposed technique is scalable in terms of runtime and memory usage (more than three orders of magnitude improvement on average). I have also utilized remainder to activate and detect Trojans. Any assignment which makes the total value of the remainder non-zero is a directed test (counter-example) that activates Trojans. I have shown that the remainder as well as directed tests can also be used to localize Trojans. Moreover, the terms and their patterns of the remainder provide valuable information to detect and correct the hidden Trojans.

The existence of a Trojan in the deeper stages of the design may make it difficult to generate the remainder. I also addressed scalability by proposing an incremental Trojan detection framework in Chapter 5. The proposed approach partitions the primary inputs' space in order to solve the security validation problem in the increasing order of the design complexity. Proposed approach can formally identify a Trojan-free IP from a Trojan-inserted one and it can automatically activate and correct hidden Trojans.

Chapter 6 described an approach for hardware Trojan localization after non-functional changes. Suppose that there is a golden model of a design (specification), and a modified version (implementation) of it (after performing some non-functional changes such as doing synthesis, adding clock trees, scan chain insertion etc.). The goal is to make sure that two versions of a design are functionally equivalent (nothing more, nothing less) and an adversary cannot hide hard-to-detect malicious modifications during design transformations. I proposed an approach based on Gröbner Basis theory. However, applying the same approach on general IPs is limited due to several reasons such as specification polynomial generation, sequential element and large number of the required unrolling for Trojan activation. In order to address these challenges, I presented a design partitioning method to generate polynomials in an efficient way and use them in our proposed algorithm to localize and detect Trojans in third-party IPs. The proposed approach is scalable and could efficiently detect and localize the hidden Trojans while industrial tools fail.

Chapter 7 addressed issues in control flow integrity measurement. To detect vulnerabilities introduced by Trojan insertion as well as CAD tools, I proposed an efficient formal analysis framework based on symbolic algebra to find FSM vulnerabilities. The proposed method tries to find inconsistencies between the specification and FSM implementation through manipulation of respective polynomials. Security properties (such as a safe transition to a protected state) are derived using specification polynomials and verified against implementation polynomials. In case of a failure, the vulnerability is reported and a counter-example is generated. I demonstrated the merit of the proposed method by detecting the vulnerabilities in various current encryption and digest FSM designs, while state-of-the-art approaches failed to identify the security flaws.

In chapter 8, I also proposed a scalable directed test generation method to activate potential hardware Trojans in RTL designs using an effective combination of concrete simulation and symbolic execution. The proposed test generation approach is scalable since it can avoid state space explosion by exploring one execution path at a time in contrast to dealing with all possible execution paths simultaneously (like conventional formal methods). I developed a threat model involving rare branches and rare assignments in RTL designs. This threat model leads to a list of potential security targets for directed test generation. To enable efficient test generation for security targets, I proposed a search heuristic which avoids traversing same branches (path segments) during traversal to improve rare branch coverage of security targets. Our experimental results demonstrated the effectiveness of our approach in activating hard-to-detect Trojans in large and complex RTL Trust-Hub benchmarks.

In Chapter 9, observability-aware test generation is discussed. Directed tests are very promising in reducing the overall validation effort since a drastically small number of directed tests are required compared to random tests to obtain the same coverage goal. Manual development of directed tests can be both error-prone and infeasible to generate all directed tests to achieve a coverage goal. In this work, I proposed a directed test generation approach which facilities the observation of expected outputs of the generated tests using the traced buffer signals. The proposed approach uses transaction-level models (TLM) for post-silicon test generation. The proposed approach has four steps: i) mapping observability constraints, ii) test mapping from lower level abstractions (e.g, RTL modules) to higher-levels of abstractions (e.g., TLM), iii) test generation using higherlevels descriptions, and iv) test translating to add details such as timing. Our experimental results using an industrial processor demonstrate significant reduction in both test-generation time and test program length compared to random or constrained-random tests.

Chapter 10 discussed about cost-effective post-silicon coverage analysis techniques. Assertions as well as a wide variety of checkers are used in pre-silicon stage to monitor certain functional scenarios. Pre-silicon checkers can be synthesized to coverage monitors in order to capture the coverage of certain events and improve the observability during post-silicon debug. Synthesizing thousands of coverage monitors can introduce unacceptable area and energy overhead. On the other hand, absence of coverage monitors would negatively impact post-silicon coverage analysis. I proposed a method to utilize on-chip

DfD infrastructure to perform post-silicon functional coverage analysis. I also evaluated the importance of each coverage monitor in order to provide a trade-off between observability versus design overhead. Our initial results show that only 10% coverage monitors (if synthesized) can provide 100% assertion coverage analysis. In contrast, if DfD architecture is not utilized, 10% synthesized monitors can provide coverage analysis for only 10% assertions.

11.1 Future Research Directions

The research presented in this dissertation can be extended in the following directions to establish trust in IoT SoCs: 1) use of analytics, formal validation, as well as side-channel analysis approaches for scalable security validation, 2) streamlined architecture for secure post-silicon and in-field control and requirements update, and 3) trustworthy synthesis of hardware designs from high-level specifications. There are several classes of hardware security vulnerabilities such as access privileges, buffer errors, resource management, information leakage, numeric errors, crypto errors and code injection as well as software and firmware attacks that threaten the security and integrity of the design. I plan to explore each class of the security vulnerabilities and generate security assertions for each type by developing a set of templates and rule-based transformation of vulnerabilities to security assertions. The generated assertions need to be mapped to the corresponding hardware/software designs and to be verified. Clearly, the research requires the knowledge of security violation patterns from the static analysis or provided by designers.

Another promising direction is to perform functional validation using machine learning. While directed tests can check for known vulnerabilities, machine learning can extend the scope for both known and unknown (e.g., known vulnerabilities with minor or major variations) SoC security vulnerabilities. Existing data in verification environment traces can be clustered into several buckets such that each bucket contains traces that have failed as the result of the same cause. Therefore, debug of known security failures can be avoided. Future research can aim for effective utilization of three widely used formal methods (equivalence checking, model checking and theorem proving) for scalable security validation. The side-channel analysis is also beneficial since it does not require activation/propagation of an unknown Trojan.

Another promising direction is to combine the advantages of logic testing and side-channel analysis for effective Trojan identification. Effective in-field debug should be performed in the case of Trojan-inserted fabricated silicon. Post-silicon validation and debug is critical for IoT applications, where the device may need to be updated during its long lifetime, or in response to changing security needs or new attacks. Future work needs to develop efficient algorithms and techniques for debugging and validating embedded devices using effective on-chip instrumentation techniques for silicon observability, penetration test generation techniques, and enabling post-silicon validation use-cases in pre-silicon validation platform. There is an inherent conflict between increasing the observability and security. Although designing effective debug infrastructures can drastically reduce the post-silicon validation and debug efforts by increasing the observability, the extra observability can facilitate security attacks such as trace buffer attack and scan-based side channel attacks.

There are several system-level security monitoring approaches that utilize dedicated hardware design to check the operation of an embedded processor instruction-by-instruction. Any deviations from the expected behavior (which may come from deign errors as well as run-time attacks) is marked as a security threat [107, 132]. I plan to propose approaches to make such security mechanisms smart and energy efficient.

Finally, existing Electronic Design Automation (EDA) tools can cause several security vulnerabilities. It is extremely important to have design tools that can check for all vulnerabilities in all phases of the design since it may be infeasible to patch later. The tools must be able to tell all possible vulnerabilities such as the existence of hardware

Trojans, information flow vulnerabilities, side channel leakage, etc., and suggest effective countermeasures.

APPENDIX A
LIST OF PUBLICATIONS

Book

1. P. Mishra and F. Farahmandi (Editors), Post-Silicon Validation and Debug, Springer, April 2018.

Book Chapters

1. Prabhat Mishra and **F. Farahmandi**, “Current SoC Design and Validation Flow,” Post-Silicon Validation and Debug, P. Mishra and F. Farahmandi (editors), Springer, 2018 (in preparation).
2. **F. Farahmandi**, “ Observability-aware Directed Test Generation,” Post-Silicon Validation and Debug, P. Mishra and F. Farahmandi (editors), Springer, 2018 (in preparation).
3. Prabhat Mishra and **F. Farahmandi**, “Putting It All Together - The Future of Post-Silicon Debug,” Post-Silicon Validation and Debug, P. Mishra and F. Farahmandi (editors), Springer, 2018 (in preparation).
4. **F. Farahmandi** , A. Ahmed, Y. Iskander and Prabhat Mishra, “Security Verification using Concolic Testing,” Security and Fault tolerance in Internet of Things, S. Chakraborty and J. Mathew (editors), Springer, 2018.
5. **F. Farahmandi**, Yuanwen Huang and Prabhat Mishra, “Formal Approaches to Hardware Trust Verification,” The Hardware Trojan War: Attacks, Myths, and Defenses, S. Bhunia and M. Tehranipoor (editors), Springer, 2017.
6. **F. Farahmandi** and Prabhat Mishra, “Validation of IP Security and Trust,” Hardware IP Security and Trust, P. Mishra, S. Bhunia and M. Tehranipoor (editors), Springer, 2017.

Journal Articles

1. **F. Farahmandi** and P. Mishra, “Automated Test Generation for Debugging Multiple Unknown Bugs in Arithmetic Circuits,” in *IEEE Transaction on Computers (TC)*, 2017 (under revision).
2. A. Nahiyan, **F. Farahmandi**, D. Forte, P. Mishra and M. Tehranipoor, “Security-aware FSM Design Flow for Mitigating Vulnerabilities to Fault Injection Attacks,” in *IEEE Transactions on Computer-Aided Design (TCAD)*, 2017 (under revision).
3. **F. Farahmandi**, Q. Xu and P. Mishra, “Survey on Observability Enhancement Methods for Post-Silicon Debug,” submitted in *Proceedings of the IEEE*, 2017.

4. **F. Farahmandi**, R. Morad, A. Ziv, Z. Nevo and P. Mishra, "Post-Silicon Functional Coverage Analysis utilizing Design-for-Debug Infrastructure," submitted to *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2017.

Conference Papers

1. **F. Farahmandi**, A. Alif, J. Cruz, Y. Iskander and P. Mishra, "Scalable Hardware Trojan Detection by Interleaving Concrete Simulation and Symbolic Execution," submitted to *ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, June 24-28, 2018.

2. A. Alif, **F. Farahmandi** and P. Mishra, "Directed Test Generation using Concolic Testing on RTL models," in *Design Automation and Test in Europe (DATE)*, Dresden, Germany, March 19-23, 2018.

3. J. Cruz, **F. Farahmandi**, A. Ahmed, and P. Mishra, "Hardware Trojan Detection using ATPG and Model Checking," in *International Conference on VLSI Design (VLSI Design)*, Pune, India, January 6 - 10, 2018.

4. **F. Farahmandi**, R. Morad, A. Ziv, Z. Nevo and P. Mishra. "Cost-Effective Analysis of Post-Silicon Functional Coverage Events," IEEE/ACM Design Automation and Test in Europe (DATE), pages 392-397, 2017.

5. **F. Farahmandi**, Y. Huang and P. Mishra. *Trojan Localization using Symbolic Algebra.* IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC), pages 591-597, 2017. **(Nominated for Best Paper Award)**

6. **F. Farahmandi** and P. Mishra. *Automated Test Generation for Debugging Arithmetic Circuits.* IEEE/ACM Design, Automation and Test in Europe Conference (DATE), pages 1351-1356, 2016.

7. **F. Farahmandi**, P. Mishra, S. Ray. *Exploiting Transaction-level Models for Observability-aware Post-Silicon Test Generation.* IEEE/ACM in Design, Automation and Test in Europe Conference (DATE), pages 1477-1480, 2016.

8. X. Guo, R. G. Dutta, Y. Jin, **F. Farahmandi**, P. Mishra. *Pre-Silicon Security Verification and Validation: A Formal Perspective.* ACM/IEEE Design Automation Conference (DAC), pages 145:1-145:6, 2015.

9. **F. Farahmandi** and P. Mishra. *Incremental Debugging of Arithmetic Circuits using Grobner Basis Reduction.* submitted to International Conference on Computer Aided Design, 2017.

10. **F. Farahmandi** and P. Mishra. *FSM Anomaly Detection using Formal Analysis.* IEEE International Conference on Computer Design , 2017.

REFERENCES

[1] https://www.synopsys.com/support/training/rtl-synthesis/design-compiler.html.

[2] J. Aarestad, D. Acharyya, R. Rad, and J. Plusquellic. Detecting trojans through leakage current analysis using multiple supply pad I_{ddq}s. In *IEEE Transactions on Information Forensics and Security*, pages 893–904, 2010.

[3] A. Adir, E. Almog, L. Fournier, E. Marcus M. Vinov, and a. Vinov. Genesys-pro: Innovations in test program generation for functional processor verification. volume 2, pages 84–93, Mar-Apr 2004.

[4] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, and A. Ziv. A unified methodology for pre-silicon verification and post-silicon validation. In *Design Automation and Test in Europe (DATE)*, pages 1590–1595, 2011.

[5] A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv. Threadmill: A post-silicon exerciser for multi-threaded processors. In *IEEE/ACM International Conference on Computer Design Automation (DAC)*, pages 860–865, 2011.

[6] A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann. Reaching coverage closure in post-silicon validation. In *International Conference on Hardware and Software: Verification and Testing (HVC)*, page 6075, 2010.

[7] B. Le, H. Mangassarian, B. Keng and A. Veneris. Non-solution implications using reverse domination in a modern sat-based debugging enviroment. In *Design Automation and Test in Europe(DATE)*, pages 629–634, 2012.

[8] Kyle Balston, Mehdi Karimibiuki, Alan J Hu, Andre Ivanov, and Steven JE Wilton. Post-silicon code coverage for multiprocessor system-on-chip designs. In *IEEE Transactions on Computers*, pages 242–246. IEEE, 2013.

[9] M. Banga and M.S. Hsiao. Trusted rtl: Trojan detection methodology in pre-silicon designs. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 56–59, 2010.

[10] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[11] K. Basu and P. Mishra. Efficient trace signal selection for post silicon validation and debug. In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pages 352 –357, jan. 2011.

[12] K. Basu and P. Mishra. Rats: Restoration-aware trace signal selection for post-silicon validation. In *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, volume 21, page 605613, 2013.

[13] Payman Behnam and Bijan Alizadeh. In-circuit mutation-based automatic correction of certain design errors using sat mechanisms. In *Test Symposium (ATS), 2015 IEEE 24th Asian*, pages 199–204. IEEE, 2015.

[14] Payman Behnam, Bijan Alizadeh, and Zainalabedin Navabi. Automatic correction of certain design errors using mutation technique. In *Test Symposium (ETS), 2014 19th IEEE European*, pages 1–2. IEEE, 2014.

[15] S. BeigMohammadi and B. Alizadeh. Combinational trace signal selection with improved state restoration for post-silicon debug. In *Design Automation and Test in Europe (DATE)*, pages 1369–1374, 2016.

[16] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE transactions on very large scale integration (VLSI) systems*, 8(3):299–316, 2000.

[17] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan. Hardware trojan attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, Aug 2014.

[18] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[19] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[20] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[21] E. Biham, Y. Carmeli, and A. Shamir. Bug attacks. *Advances in Cryptology*, page 221240, 2008.

[22] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. *Journal of CRYPTOLOGY*, 4(1):3–72, 1991.

[23] M. Boule, J. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *International Conference on Computer Design (ICCD)*, pages 294–299, 2006.

[24] M. Boulè and Z. Zilic. Automata-based assertion-checker synthesis of psl properties. In *ACM Transaction Design Autom. Electr. Syst.*, volume 13(1), 2008.

[25] Randal E Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 535–541. ACM, 1995.

[26] B. Buchberger. A criterion for detecting unnecessary reductions in the construction of a groebner bases. In *EUROSAM*, 1979.

[27] Bruno Buchberger. Some properties of gröbner-bases for polynomial ideals. *ACM SIGSAM Bulletin*, 10(4):19–24, 1976.

[28] Cadence Berkeley Lab, Available at http://www.kenmcmil.com. *The Cadence SMV Model Checker*.

[29] Burçin Çakir and Sharad Malik. Hardware trojan detection for gate-level ics using signal correlation based clustering. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 471–476. EDA Consortium, 2015.

[30] R. S. Chakraborty, F. Wolf, C. Papachristou, and S. Bhunia. Mero: A statistical approach for hardware trojan detection. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES'09)*, pages 369–410, 2009.

[31] M. Chen, P. Mishra, and D. Kalita. Automatic RTL Test Generation from SystemC TLM Specifications. volume 11, July 2012.

[32] M. Chen, X. Qin, H. Koo, and P. Mishra. *System-level Validation - high-level modeling and directed test generation techniques*. Springer, 2012.

[33] Mingsong Chen and Prabhat Mishra. Functional test generation using efficient property clustering and learning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):396–404, 2010.

[34] Maciej Ciesielski, Cunxi Yu, Walter Brown, Duo Liu, and André Rossi. Verification of gate-level arithmetic circuits by function extraction. In *Proceedings of the 52nd Annual Design Automation Conference*, page 52. ACM, 2015.

[35] Maciej J Ciesielski, Priyank Kalla, Zhihong Zheng, and Bruno Rouzeyre. Taylor expansion diagrams: A compact, canonical representation with applications to symbolic verification. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 285–289. IEEE, 2002.

[36] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[37] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[38] D. Cox, J. Little, and D. O'Shea. *Ideals, varieties, and algorithms*. Springer, 1997.

[39] D. Cox, J. little, and D. O'shea. Ideal, varieties and algorithm: An introduction to computational algebraic geometry and commutative algebra. In *Springer*, 2007.

[40] David Cox, John Little, and Donal O'shea. *Ideals, varieties, and algorithms*, volume 3. Springer, 1992.

[41] Jonathan Cruz, Farimah Farahmandi, Alif Ahmed, and Prabhat Mishra. Hardware trojan detection using atpg and model checking. In *VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018 31st International Conference on*, pages 91–96. IEEE, 2018.

[42] David Cyrluk, Sreeranga Rajan, Natarajan Shankar, and Mandayam Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design*, pages 203–222. Springer, 1995.

[43] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[44] Carson Dunbar and Gang Qu. Designing trusted embedded systems from finite state machines. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):153, 2014.

[45] Bruno Dutertre. Yices 2.2. In *CAV*, pages 737–744, 2014.

[46] F. Farahmandi and B. Alizadeh. Grobner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction. In *Microprocessor and Microsystems - Embedded Hardware Design*, pages 83–96, 2015.

[47] F. Farahmandi and 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*, pages 83–96, 2015.

[48] F. Farahmandi, P. Mishra, and S. Ray. Exploiting transaction level models for observability-aware post-silicon test generation. In *Design Automation and Test in Europe (DATE)*, pages 1477–1480, 2016.

[49] Farimah Farahmandi and Bijan Alizadeh. Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction. *Microprocessors and Microsystems*, 39(2):83–96, 2015.

[50] Farimah Farahmandi, Bijan Alizadeh, and Zain Navabi. Effective combination of algebraic techniques and decision diagrams to formally verify large arithmetic circuits. In *2014 IEEE Computer Society Annual Symposium on VLSI*, pages 338–343. IEEE, 2014.

[51] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. Trojan localization using symbolic algebra. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 591–597. IEEE, 2017.

[52] Farimah Farahmandi and Prabhat Mishra. Automated test generation for debugging arithmetic circuits. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1351–1356. IEEE, 2016.

[53] Farimah Farahmandi, Ronny Morad, Avi Ziv, Ziv Nevo, and Prabhat Mishra. Cost-effective analysis of post-silicon functional coverage events. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 392–397. IEEE, 2017.

[54] Nicole Fern and Kwang-Ting Tim Cheng. Detecting hardware trojans in unspecified functionality using mutation testing. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 560–566. IEEE Press, 2015.

[55] Nicole Fern, Shrikant Kulkarni, and Kwang-Ting Tim Cheng. Hardware trojans hidden in RTL don't cares – Automated insertion and prevention methodologies. In *Test Conference (ITC), 2015 IEEE International*, pages 1–8. IEEE, 2015.

[56] Nicole Fern, Ismail San, and Kwang-Ting Tim Cheng. Detecting hardware trojans in unspecified functionality through solving satisfiability problems. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 598–504. IEEE, 2017.

[57] Nicole Fern, Ismail San, Cetin Kaya Koç, and Kwang-Ting Tim Cheng. Hardware trojans in incompletely specified on-chip bus systems. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 527–530. EDA Consortium, 2016.

[58] Samaneh Ghandali, Cunxi Yu, Duo Liu, Walter Brown, and Maciej Ciesielski. Logic debugging of arithmetic circuits. In *2015 IEEE Computer Society Annual Symposium on VLSI*, pages 113–118. IEEE, 2015.

[59] Godefroid et al. DART: directed automated random testing. In *SIGPLAN*, pages 213–223, 2005.

[60] Lawrence H Goldstein and Evelyn L Thigpen. Scoap: Sandia controllability/observability analysis program. In *Proceedings of the 17th Design Automation Conference*, pages 190–196. ACM, 1980.

[61] Pfister G. Schnemann Greuel, G.-M. *2012. SINGULAR 3.1.3 A Computer Algebra System for Polynomial Computations. Centre for Computer Algebra.* http://www.singular.uni-kl.de.

[62] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra. Pre-silicon security verification and validation: A formal perspective. In *ACM/IEEE Design Automation Conference (DAC)*, 2015.

[63] Xiaolong Guo, Raj Gautam Dutta, Prabhat Mishra, and Yier Jin. Scalable soc trust verification using integrated theorem proving and model checking.

[64] Syed Rafay Hasan, Charles A Kamhoua, Kevin A Kwiat, and Laurent Njilla. Translating circuit behavior manifestations of hardware trojans using model checkers into run-time trojan detection monitors. In *Hardware-Oriented Security and Trust (AsianHOST), IEEE Asian*, pages 1–6. IEEE, 2016.

[65] Matthew Hicks, Murph Finnicum, Samuel T King, Milo MK Martin, and Jonathan M Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 159–172. IEEE, 2010.

[66] http://opencores.org. *OpenCores*.

[67] https://cwe.mitre.org. *Common Weakness Enumeration*.

[68] https://www.fbo.gov/index?s=opportunity&mode=form&id=ea2550cb0c42eb91c7292377824a58b7. *DARPA System Security Integrated Through Hardware and Firmware (SSITH)*.

[69] https://www.trust-hub.org/. *Trust-HUB*.

[70] http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Test/Pages/TetraMAXATPG.aspx. *Synopsys, Tetramax ATPG*.

[71] http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/Formality.aspx. *Synopsys, Formality*, 2015.

[72] http://www.vlsiip.com/formality/ug.pdf. *Formality, User Guide*, 2007.

[73] Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. Detecting hardware trojans with gate-level information-flow tracking. *Computer*, 49(8):44–52, 2016.

[74] Yuanwen Huang, Swarup Bhunia, and Prabhat Mishra. Mers: Statistical test generation for side-channel analysis based trojan detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 130–141. ACM, 2016.

[75] Ismari et al. On detecting delay anomalies introduced by hardware trojans. In *ICCAD*, pages 1–7, 2016.

[76] Y. Jin and Y. Makris. Hardware trojan detection using path delay fingerprint. In *Hardware-Oriented Security and Trust (HOST)*, pages 51–57, 2008.

[77] Y. Jin and Y. Makris. Proof carrying-based information flow tracking for data secrecy protection and hardware trust. In *VLSI Test Symposium (VTS)*, pages 252–257, 2012.

[78] Yier Jin. EDA tools trust evaluation through security property proofs. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–4, 2014.

[79] Yier Jin, Bo Yang, and Yiorgos Makris. Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 99–106, 2013.

[80] Wisam Kadry, Dimtry Krestyashyn, Arkadiy Morgenshtein, Amir Nahir, Vitali Sokhin, Jin Sung Park, Sung-Boem Park, Wookyeong Jeong, and Jae Cheol Son. Comparative study of test generation methods for simulation accelerators. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 321–324. IEEE, 2015.

[81] Ramesh Karri, Jeyavijayan Rajendran, Kurt Roseland, and Mohammad Tehranipoor. Trustworthy hardware: Identifying and classifying hardware trojans. In *IEEE Computer*, pages 39–46, 2010.

[82] Brian Keng and Andreas Veneris. Path-directed abstraction and refinement for sat-based design debugging. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(10):1609–1622, 2013.

[83] Ho Fai Ko and N. Nicolici. Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(2):285 –297, feb. 2009.

[84] C. Koc and T. Acar. Montgomery multiplication in $GF(2^k)$. In *Designs, Codes and Cryptography*, volume 14, pages 57–69, 1998.

[85] Heon-Mo Koo and Prabhat Mishra. Test generation using sat-based bounded model checking for validation of pipelined processors. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pages 362–365. ACM, 2006.

[86] Kroening et al. *EBMC*. http://www.cprover.org/ebmc.

[87] M. Li and A. Davoodi. A hybrid approach for fast and accurate trace signal selection for post-silicon debug. In *Design Automation and Test in Europe (DATE)*, page 485490, 2013.

[88] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan. Towards coverage closure: Using goldmine assertions for generating design validation stimulus. In *Design Automation and Test in Europe (DATE)*, pages 173–178, 2011.

[89] Lingyi Liu and Shobha Vasudevan. Efficient validation input generation in rtl by hybridized source code analysis. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[90] Liu et al. Hardware trojan detection through golden chip-free statistical side-channel fingerprinting. In *DAC*, pages 1–6, 2014.

[91] J. Lv, P. Kalla, and F. Enescu. Efficient groebner basis reductions for formal verification of galois field multipliers. In *Proceedings Design, Automation and Test in Europe Conf. (DATE)*, pages 899–904, 2012.

[92] J. Lv, P. Kalla, and F. Enescu. Efficient groebner basis reductions for formal verification of galois field arithmetic circuits. In *IEEE Transactions on CAD (TCAD)*, volume 32, pages 1409 – 1420, 2013.

[93] M. J. Ciesielski, C. Yu, W. Brown, D. Liu and A. Rossi. Verification of gate-level arithmetic circuits by function extraction. In *IEEE/ACM International Conference on Computer Design Automation(DAC)*, pages 1–6, 2015.

[94] and J. Fan M. Knežević, and K. Sakiyama and I. Verbauwhed. Modular reduction in $GF(2^n)$ without pre-computational phase. In *Proceedings of the International Workshop on Arithmetic of Finite Fields*, pages 77–87, 2008.

[95] S. Ma, D. Pal, R. Jiang, S. Ray, and S. Vasudevan. Can't see the forest for the trees: State restoration's limitations in post-silicon trace signal selection. In *International Conference On Computer Aided Design (ICCAD)*, pages 146:1–146:6, 2015.

[96] Siyad C Ma, Piero Franco, and Edward J McCluskey. An experimental chip to evaluate test techniques experiment results. In *Test Conference, 1995. Proceedings., International*, pages 663–672. IEEE, 1995.

[97] Hratch Mangassarian, Andreas Veneris, Sean Safarpour, Marco Benedetti, and Duncan Smith. A performance-driven qbf-based iterative logic array representation with applications to verification, debug and test. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 240–245. IEEE, 2007.

[98] Takeshi Matsumoto, Shohei Ono, and Masahiro Fujita. An efficient method to localize and correct bugs in high-level designs using counterexamples and potential dependence. In *VLSI and System-on-Chip, 2012 (VLSI-SoC), IEEE/IFIP 20th International Conference on*, pages 291–294. IEEE, 2012.

[99] Kenneth L McMillan. *Model checking.* John Wiley and Sons Ltd., 2003.

[100] Travis Meade, Shaojie Zhang, and Yier Jin. Netlist reverse engineering for high-level functionality reconstruction. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 655–660. IEEE, 2016.

[101] S. Mitra, S. A. Seshia, and N. Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *IEEE/ACM International Conference on Computer Design Automation (DAC)*, pages 12–17, June 2010.

[102] Adib Nahiyan, Kan Xiao, Kun Yang, Yier Jin, Domenic Forte, and Mark Tehranipoor. Avfsm: a framework for identifying and mitigating vulnerabilities in fsms. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2016.

[103] Seetharam Narasimhan, Dongdong Du, Rajat Subhra Chakraborty, Somnath Paul, Francis G Wolff, Christos A Papachristou, Kaushik Roy, and Swarup Bhunia.

Hardware trojan detection by multiple-parameter side-channel analysis. *IEEE Transactions on computers*, 62(11):2183–2195, 2013.

[104] George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM, 1997.

[105] Masaru Oya, Youhua Shi, Masao Yanagisawa, and Nozomu Togawa. A score-based classification method for identifying hardware-trojans at gate-level netlists. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 465–470, 2015.

[106] P. Patra. On the cusp of a validation wall. In *IEEE Design and. Test of Computers*, volume 24, pages 193–196, 2007.

[107] Arman Pouraghily, Tilman Wolf, and Russell Tessier. Hardware support for embedded operating system security. In *Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International Conference on*, pages 61–66. IEEE, 2017.

[108] Xiaoke Qin and Prabhat Mishra. Scalable test generation by interleaving concrete and symbolic execution. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*, pages 104–109. IEEE, 2014.

[109] Yingxin Qiu, Huawei Li, Tiancheng Wang, Bo Liu, Yingke Gao, and Xiaowei Li. Property coverage analysis based trustworthiness verification for potential threats from eda tools. In *Asian Test Symposium (ATS), 2016 IEEE 25th*, pages 43–48. IEEE, 2016.

[110] R. E. Bryant. Graph-based algorithms for boolean function manipulation,. In *IEEE Transactions on Computers*, pages 677–691, 1986.

[111] K. Rahmani, P. Mishra, and S. Ray. Scalable trace signal selection using machine learning. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 384–389, Oct 2013.

[112] Jeyavijayan Rajendran, Arunshankar Muruga Dhandayuthapany, Vivekananda Vedula, and Ramesh Karri. Formal security verification of third party intellectual property cores for information leakage. In *VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID), 2016 29th International Conference on*, pages 547–552. IEEE, 2016.

[113] Jeyavijayan Rajendran, Vivekananda Vedula, and Ramesh Karri. Detecting malicious modifications of data in third-party intellectual property cores. In *Proceedings of the 52nd Annual Design Automation Conference*, page 112. ACM, 2015.

[114] Michael Rathmair and Florian Schupfer. Hardware trojan detection by specifying malicious circuit properties. In *Electronics Information and Emergency Communication (ICEIEC), 2013 IEEE 4th International Conference on*, pages 317–320. IEEE, 2013.

[115] Sandip Ray, Yier Jin, and Arijit Raychowdhury. The changing computing paradigm with internet of things: A tutorial introduction. *IEEE Design & Test*, 33(2):76–96, 2016.

[116] S. Mirzaeian, F. Zheng and K. T. Chen. Rtl error diagnosis using a word-level sat-solver. In *Proc. IEEE Int. Test Conference (ITC)*, pages 1–8, 2008.

[117] Sean Safarpour and Andreas Veneris. Abstraction and refinement techniques in automated design debugging. In *Seventh International Workshop on Microprocessor Test and Verification (MTV'06)*, pages 88–93. IEEE, 2006.

[118] S. Saha, R. Chakraborty, S.S. Nuthakki, Anshul, and D. Mukhopadhyay. Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 577–596, 2015.

[119] Hassan Salmani. Cotd: Reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist. *IEEE Transactions on Information Forensics and Security*, 12(2):338–350, 2017.

[120] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. A novel technique for improving hardware trojan detection and reducing trojan activation time. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(1):112–125, 2012.

[121] Amr Sayed-Ahmed, Daniel Gro, Mathias Soeken, Rolf Drechsler, et al. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1048–1053. IEEE, 2016.

[122] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[123] Sen et al. CUTE: a concolic unit testing engine for C. In *SIGSOFT*, pages 263–272, 2005.

[124] Alexander Smith, Andreas Veneris, Moayad Fahim Ali, and Anastasios Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1606–1621, 2005.

[125] Cynthia Sturton, Matthew Hicks, David Wagner, and Samuel T King. Defeating uci: Building stealthy and malicious hardware. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 64–77. IEEE, 2011.

[126] Andre Sülflow, Görschwin Fey, and Rolf Drechsler. Experimental studies on smt-based debugging. In *IEEE Workshop on RTL and High Level Testing*, pages 93–98, 2008.

[127] Xiaojun Sun, Priyank Kalla, and Florian Enescu. Word-level traversal of finite state machines using algebraic geometry. In *High Level Design Validation and Test Workshop (HLDVT), 2016 IEEE International*, pages 142–149. IEEE, 2016.

[128] Xiaojun Sun, Priyank Kalla, Tim Pruss, and Florian Enescu. Formal verification of sequential galois field arithmetic circuits using algebraic geometry. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1623–1628. EDA Consortium, 2015.

[129] Berk Sunar, Gunnar Gaubatz, and Erkay Savas. Sequential circuit design for embedded cryptographic applications resilient to adversarial faults. *IEEE Transactions on Computers*, 57(1):126–138, 2008.

[130] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design and Test of Computers*, 27(1):10–25, 2010.

[131] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[132] Tedy Thomas, Arman Pouraghily, Kekai Hu, Russell Tessier, and Tilman Wolf. Multi-task support for security-enabled embedded processors. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 136–143. IEEE, 2015.

[133] Shobha Vasudevan, E Allen Emerson, and Jacob A Abraham. Efficient model checking of hardware using conditioned slicing. *Electronic Notes in Theoretical Computer Science*, 128(6):279–294, 2005.

[134] A. Waksman, M. Suozzo, and S. Sethumadhavan. Fanci: Identification of stealthy malicious logic using boolean functional analysis. In *ACM SIGSAC Conference on Computer &#38; Communications Security*, pages 697–708, 2013.

[135] Waksman et al. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *CCS*, pages 697–708, 2013.

[136] Zhen Wang and Mark Karpovsky. Robust fsms for cryptographic devices resilient to strong fault injection attacks. In *2010 IEEE 16th International On-Line Testing Symposium*, pages 240–245. IEEE, 2010.

[137] Wang et al. Hardware trojan detection and isolation using current integration and localized current analysis. In *DFT*, pages 87–95, 2008.

[138] O. Wienand, M. Welder, D. Stoffel, W. Kunz, and G. M. Greuel. An algebraic approach for proving data correctness in arithmetic data paths. In *Computer Aided Verification (CAV)*, pages 473–486, 2008.

[139] S. Williams. *Icarus verilog.* On-line: http://iverilog.icarus.com.

[140] S. Yerramilli. Addressing post-silicon validation challenge: Leverage validation and test synergy. In *Keynote, Intl. Test Conf.*, 2006.

[141] Bilgiday Yuce, Nahid F Ghalaty, Chinmay Deshpande, Conor Patrick, Leyla Nazhandali, and Patrick Schaumont. Fame: Fault-attack aware microprocessor extensions for hardware fault detection and software fault response. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, page 8. ACM, 2016.

[142] Jie Zhang, Feng Yuan, Linxiao Wei, Yannan Liu, and Qiang Xu. Veritrust: verification for hardware trust. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(7):1148–1161, 2015.

[143] Jie Zhang, Feng Yuan, and Qiang 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*, pages 153–166. ACM, 2014.

[144] Xuehui Zhang and Mohammad Tehranipoor. Case study: Detecting hardware trojans in third-party digital ip cores. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 67–70. IEEE, 2011.

BIOGRAPHICAL SKETCH

Farimah Farahmandi is pursuing Ph.D. in Department of Computer and Information Science and Engineering (CISE) at the University of Florida. She received her B.S. and M.S. from the Department of Electrical and Computer Engineering (ECE), the University of Tehran, Iran in 2010 and 2013, respectively. In August 2013, she started pursuing her Ph.D. under the supervision of Prof. Prabhat Mishra. Her research is focused on developing analytical models and computational methods for design and verification of secure, trustworthy and energy-efficient systems. Her research has resulted in one book, six book chapters, and thirteen publications in premier ACM/IEEE journals and conferences including Design Automation Conference (DAC) and Design, Automation and Test in Europe (DATE). Her research has been recognized by several awards including IEEE System Validation and Debug Technology Committee Student Research Award, Gartner Group Info-Tech Scholarship, nomination for Best Paper Award in ASPDAC 2017, and DAC Richard Newton Young Student Fellowship. She was a research intern in advanced security research group at Cisco in summer 2016. She has actively collaborated with various research groups (IBM, NXP, Intel, and Cisco) that led to several joint publications. She is the lead researcher of several projects funded by Cisco, National Science Foundation and Semiconductor Research Corporation. She also has received several travel grant awards from Design Automation Conference (2015), ACM Capital Region Celebration of Women in Computing Conference Scholarship (2017) and Microsoft Research and ACM Full Scholarship CRA-W Grad Workshop. She served as a reviewer for multiple top-tier academic conferences and journals.