TEST GENERATION FOR SYSTEM-ON-CHIP SECURITY VALIDATION

By

YANGDI LYU

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

2020

© 2020 Yangdi Lyu

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere appreciation to my advisor, Prof. Prabhat Mishra, who provided the persistent support and guidance for my Ph.D. study. His patience, motivation and immense knowledge helped me in all aspects of research and writing this dissertation. He is the person who made my Ph.D. research and this dissertation come true.

Besides my advisor, I would like to thank the rest of my Ph.D. committee members (Prof. Sartaj Sahni, Prof. My Thai, and Prof. Swarup Bhunia) for their constructive recommendations and insightful critiques. Their diverse expertise and knowledge helped me improve the quality of my research and this dissertation.

I thank my fellow labmates: Yuanwen Huang, Farimah Farahmandi, Subodha Charles, Alif Ahmed, Zhixin Pan, Daniel Volya, Hasini Witharana, and Jonathan Cruz. It was my great pleasure to collaborate with them.

Last but not least, I sincerely acknowledge the support and great love of my parents, my wife and my son. This dissertation would not be possible without their unconditional support and love. I would like to offer my special appreciation to my wife, Xiaojie, who contributes a lot to the family and encouraged me to make the decision to start my Ph.D. Our happy family and the beautiful UF campus have made the five-year doctoral journey so delightful. I dedicate this dissertation to them.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS . . . 3

LIST OF TABLES . . . 8

LIST OF FIGURES . . . 9

ABSTRACT . . . 12

CHAPTER

1 INTRODUCTION . . . 13

1.1 SoC Validation Methods . . . 17
1.1.1 Formal Methods . . . 17
1.1.2 Simulation-based Validation . . . 18
1.1.3 Side-channel Analysis . . . 18
1.2 SoC Security Validation Challenges . . . 19
1.2.1 A Wide Variety of Vulnerabilities . . . 20
1.2.2 Controllability and Observability . . . 20
1.2.3 Lack of Effective and Scalable Validation Techniques . . . 21
1.3 Research Contributions . . . 25
1.4 Dissertation Organization . . . 28

2 BACKGROUND AND RELATED WORK . . . 30

2.1 SoC Security Validation using Formal Methods . . . 30
2.2 SoC Security Validation using Simulation-based Validation . . . 30
2.2.1 Random/Constrained-Random Simulation . . . 31
2.2.2 Directed Test Generation using Formal Methods . . . 31
2.2.3 Statistical Methods . . . 32
2.2.4 Concolic Testing . . . 35
2.2.4.1 Concolic Testing of Software Designs . . . 36
2.2.4.2 Concolic Testing of Hardware Designs . . . 36
2.3 SoC Security Validation using Side-channel Analysis . . . 37
2.3.1 Dynamic Current based Side-Channel Analysis . . . 37
2.3.2 Path Delay based Side-Channel Analysis . . . 39
2.4 Summary . . . 40

3 SYSTEM-ON-CHIP SECURITY ASSERTIONS . . . 41

3.1 Assertion-based Validation . . . 42
3.1.1 Assertion Languages . . . 43
3.1.2 Automated Assertion Generation . . . 43
3.2 SoC Security Vulnerabilities . . . 44
3.2.1 Permissions and Privileges . . . 44

3.2.2 Resource Management . . . 45
3.2.3 Illegal States and Transitions . . . 45
3.2.4 Buffer Issues . . . 45
3.2.5 Information Leakage . . . 45
3.2.6 Numeric Exceptions . . . 46
3.2.7 Malicious Implants . . . 46
3.3 SoC Security Assertions . . . 46
3.3.1 Embedding of Security Assertions . . . 46
3.3.2 Generation of Security Assertions . . . 47
3.4 Case Studies . . . 49
3.4.1 Arbiter . . . 50
3.4.2 PCI . . . 51
3.4.3 USB Protocol . . . 51
3.4.4 A Simplified Memory Design . . . 52
3.4.5 Gaussian Noise Generator (GNG) . . . 54
3.4.6 AES . . . 55
3.5 Summary . . . 56

4 SCALABLE CONCOLIC TESTING OF RTL MODELS . . . 58

4.1 Overview and Problem Formulation . . . 59
4.1.1 Modeling of Targets . . . 60
4.1.2 Overview . . . 60
4.2 Test Generation using Concolic Testing . . . 61
4.2.1 RTL Code Instrumentation . . . 61
4.2.2 Contribution-aware Edge Realignment . . . 63
4.2.3 Distance Computation . . . 67
4.2.4 Path Exploration . . . 69
4.2.4.1 Dynamic Distance Update . . . 71
4.3 Optimizations for Covering Multiple Targets . . . 72
4.3.1 Target Pruning . . . 73
4.3.2 Target Clustering . . . 75
4.4 Experiments . . . 77
4.4.1 Experimental Setup . . . 77
4.4.2 Performance Comparison . . . 77
4.4.3 Scalability Comparison . . . 80
4.4.4 Effect of Target Pruning . . . 82
4.4.5 Effect of Edge Realignment . . . 84
4.5 Summary . . . 84

5 TEST GENERATION FOR ACTIVATION OF ASSERTIONS . . . 86

5.1 Problem Formulation . . . 88
5.2 Conversion of Assertions to Branches . . . 89
5.2.1 Simplified Abstract Syntax Tree . . . 90
5.2.2 Adjust AST with Timing . . . 91

5.2.3 Conversion of AST to Branch Target . . . 92
5.2.4 Complexity Analysis . . . 93
5.3 Test Generation using Concolic Testing . . . 94
5.3.1 Overview . . . 94
5.3.2 Selection of Alternate Branches in CFG . . . 95
5.4 Experiments . . . 96
5.4.1 Experimental Setup . . . 96
5.4.2 Benchmarks and Assertions . . . 97
5.4.3 Test Generation Results . . . 97
5.5 Summary . . . 99

6 TEST GENERATION FOR VALIDATION OF CACHE COHERENCE PROTOCOLS 100

6.1 Background . . . 101
6.2 Test Generation for Validation of Cache Coherence Protocols . . . 102
6.3 Scalable Test Generation using Quotient Space . . . 103
6.4 Experiments . . . 109
6.4.1 Experimental Setup . . . 109
6.4.2 Test Generation for Quotient Protocol . . . 112
6.5 Summary . . . 114

7 SCALABLE ACTIVATION OF RARE TRIGGERS . . . 115

7.1 Motivation . . . 116
7.1.1 Maximal Clique Problem . . . 118
7.2 Scalable Activation of Rare Triggers . . . 120
7.2.1 Definition and Notations . . . 120
7.2.2 Mapping Trigger Activation to Clique Cover Problem . . . 122
7.2.3 Directed Test Generation Scheme . . . 123
7.2.4 Test Generation Algorithms . . . 125
7.2.4.1 Test Generation using Clique Enumeration . . . 125
7.2.4.2 Efficient Test Generation using Clique Sampling and Lazy Construction . . . 127
7.2.5 Scalable TRAMAC by Parallelization of Clique Sampling . . . 128
7.2.6 Effectiveness of Random Clique Sampling . . . 129
7.3 Experiments . . . 132
7.3.1 Experimental Setup . . . 132
7.3.2 The Effects of Trigger Points . . . 133
7.3.3 Performance Evaluation . . . 134
7.3.4 Parallelism Evaluation . . . 138
7.3.5 Compactness and Efficiency . . . 139
7.3.6 Trigger Coverage . . . 141
7.4 Summary . . . 143

8 TROJAN DETECTION USING CURRENT-BASED SIDE-CHANNEL ANALYSIS . . 145
8.1 Problem Formulation and Motivation . . . 146
8.1.1 Problem Formulation . . . 146
8.1.2 An Illustrative Example . . . 147
8.1.3 Motivation and Research Challenges . . . 148
8.2 Generation of Effective Test Patterns . . . 149
8.2.1 Generation of the First Patterns . . . 150
8.2.2 Searching for the Best Succeeding Pattern . . . 151
8.2.2.1 Initialization . . . 153
8.2.2.2 Fitness Computation . . . 154
8.2.2.3 Selection . . . 154
8.2.2.4 Crossover and Mutation . . . 154
8.2.3 Selection of $TriggerLimit$ . . . 155
8.3 Experiments . . . 156
8.3.1 Experimental Setup . . . 156
8.3.2 Generation of Hardware Trojans . . . 157
8.3.3 Performance Evaluation . . . 158
8.3.3.1 Sensitivity comparison . . . 160
8.3.3.2 Detected Trojans . . . 162
8.3.3.3 Test generation time . . . 163
8.3.4 Evaluation of Original Switching . . . 163
8.3.5 Concurrency of MaxSense . . . 166
8.4 Summary . . . 167
9 TROJAN DETECTION USING DELAY-BASED SIDE-CHANNEL ANALYSIS 168
9.1 Test Generation for Path Delay Analysis . . . 169
9.1.1 Test Generation for Path Delay Maximization . . . 170
9.1.2 Hamming-distance based Reordering . . . 172
9.2 Experimental Results . . . 174
9.2.1 Experimental Setup . . . 174
9.2.2 Path Delay Computation . . . 175
9.2.3 Evaluation Criteria . . . 176
9.2.4 Statistical Evaluation . . . 176
9.3 Summary . . . 180
10 CONCLUSIONS AND FUTURE WORK . . . 181
10.1 Conclusions . . . 181
10.2 Future Research Directions . . . 184
APPENDIX: LIST OF PUBLICATIONS . . . 186
REFERENCES . . . 188
BIOGRAPHICAL SKETCH . . . 198

LIST OF TABLES

Table page

1-1 Seven classes of SoC security vulnerabilities . . . 20

3-1 Commonly used temporal operators in LTL [1] . . . 43

3-2 Types of vulnerabilities explored in the six benchmarks. . . . 50

4-1 The results of satisfiability checking in line 10 of Algorithm 1 for the target BB7. . . 67

4-2 Comparison of target coverage using [2], [3] and our approach on 20 targets. . . . 79

4-3 Comparison of memory requirement using EBMC and our approach on one target. . 81

4-4 The number of iterations that each block is selected as the best alternative block in exploring paths for Listing 4.1. . . . 84

5-1 Performance comparison of our approach with EBMC [4] in activating assertions. . . 98

6-1 Gem5 simulation parameters . . . 110

7-1 Comparison of TARMAC with random simulation and MERO for trigger activation coverage over 1000 randomly sampled 8-trigger conditions. . . . 136

7-2 Comparison of TARMAC with random simulation and MERO for trigger activation coverage over 1000 randomly sampled 8-trigger conditions. . . . 137

8-1 Comparison of MaxSense with NDT+GA [5] and MERS-s [6] over 1000 Trojans. . . 159

9-1 Performance comparison of our approach with random simulation and ATPG over 1000 randomly sampled Trojans. . . . 178

9-2 Test generation time of our approach in all benchmarks. . . . 179

LIST OF FIGURES

Figure		page
1-1	Various applications of IoT devices. [7]	13
1-2	An SoC design integrates a wide variety of IPs in a chip.	14
1-3	Among the recorded vulnerabilities in 2015 from MITRE/NIST CVE website, 43% were software-assisted hardware vulnerabilities. [8]	15
1-4	The impact of vulnerabilities and the hardness of patching increase from software, firmware to hardware.	15
1-5	Hardware vulnerabilities come from careless designers and buggy EDA tools in hardware design, and untrusted foundries in manufacturing.	16
1-6	The overview of my research.	17
1-7	Four major categories of formal verification methods. [9]	18
1-8	Simulation-based validation.	18
1-9	Side-channel analysis.	19
1-10	Poor observability in hardware validation. [10]	21
1-11	Full system FSM of three processor MESI-based system. [11]	23
1-12	An example hardware Trojan.	24
1-13	Random simulation cannot guarantee the coverage of all targets.	25
1-14	Dissertation outline.	29
2-1	The frameworks of MERO [12] and MERS [6].	33
2-2	The main steps of concolic testing.	34
2-3	Alternative branch selection in concolic testing.	35
2-4	Side-channel analysis detects hardware Trojans by comparing the difference of side channel signatures.	38
2-5	Two types of impact on path delay from hardware Trojans [13].	39
3-1	The framework for defining and utilizing SoC security assertions.	42
3-2	Overview of our assertion generation framework for different classes of vulnerabilities.	46
3-3	Comparison of detected vulnerabilities by our assertions and Goldmine [14].	49
3-4	A simple arbiter with four inputs (clk not shown) and two outputs.	50

4-1 Comparison of four approaches in covering two targets T_1 and T_2. 59

4-2 The overview of our test generation framework using concolic testing. 60

4-3 Comparison of edge realignment by our approach and [3]. 63

4-4 The distance between a basic block and the target in realigned CFGs. 69

4-5 CFG for the design in Listing 4.2. T0, T1, and T2 represent three targets. 73

4-6 Two simulation paths for the design in Listing 4.2 (unrolled for three cycles). 75

4-7 The comparison of memory requirements of our approach and EBMC. 82

4-8 The number of targets that are pruned. 83

5-1 Our approach converts assertions to branch targets and activates them non-vacuously. 87

5-2 AST adjustment with timing. Logic operator, implication and delay are non-terminal nodes (oval), and others are terminals (rectangle). 91

5-3 Overview of our framework to activate the branches converted from assertions. . . . 94

5-4 Chaining of related blocks in CFGs to assist alternative branch selection. 95

5-5 Memory requirement with respect to the total lines of code in custom benchmarks. . 99

6-1 State transitions for a cache block in MSI protocol. 101

6-2 The decomposition of the state space of SI protocol with 3 cores. 103

6-3 The tests generated by Euler traversal of the upper right sub-structure of hypercube. 103

6-4 The original state space and its corresponding quotient space of SI with 3 cores. . . 106

6-5 Complexity of quotient protocol with respect to number of orbits α. 107

6-6 Evaluation framework of our experiment. 111

6-7 Test generation time (left y-axis) and coverage (right y-axis) in the original space (MESI with 32 cores) of PMESI protocol with different number of orbits. 112

6-8 Total cost vs. number of orbits (α) for PMESI protocol with 64 cores. 113

6-9 Transition coverage vs. time cost for PMESI protocol with 64 cores and 15 orbits. . 114

7-1 A simple combinational Trojan with 3 triggers. 116

7-2 The number of times each rare signal is activated by by MERO. 119

7-3 The percentage of rare signals that are activated at least N times by MERO. 119

7-4 Overview of our proposed (TARMAC) paradigm. 120

7-5 A hardware Trojan with a trigger condition constructed by three rare signals. 121

7-6 The satisfiability graph with 4 PTS (A,B,C,D) from Figure 7-5, with logic expressions and rare values in parentheses. 122

7-7 The relative size of trigger conditions compared to maximal SAT cliques. 131

7-8 Experimental setup for evaluation of TARMAC compared to $N-$detect approach. . 133

7-9 Trigger condition coverage of TARMAC and MERO on c2670 and MIPS with respect to the number of test vectors and the number of trigger points. 135

7-10 The time of Algorithm 8 applied in MIPS with different number of threads. 139

7-11 Trigger coverage with respect to the number of test vectors. 140

7-12 The distribution of rare signal hits by the generated test set in all benchmarks. . . . 142

8-1 The two objectives to maximize the sensitivity in current-based side-channel analysis. 146

8-2 The example of a Trojan inserted into c17. 147

8-3 The overview of our framework MaxSense. 149

8-4 The first iteration of GA for generating the best second pattern for $u = 11100$. . . . 153

8-5 The average sensitivity of two benchmarks with respect to the length of tests. . . . 161

8-6 The distributions of sensitivities by three approaches over 1000 Trojans. 162

8-7 The distribution of the original switching in the golden design. 164

8-8 Hamming distance of all pairs of test patterns by MaxSense. 165

8-9 The test generation time of MaxSense with multi-core platforms for MIPS processor. 166

9-1 Path delay measurement using shadow registers [15]. 169

9-2 The small delay difference by existing approaches with the same critical path. 171

9-3 Our approach maximizes delay difference by changing critical paths. 171

9-4 The constraints to ensure a critical path from the trigger to the output layer. 173

9-5 A longer path may mask the delay from the Trojan. 174

9-6 The number of detected Trojan given the noise of $\pm 7.5\%$ noise. 177

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

TEST GENERATION FOR SYSTEM-ON-CHIP SECURITY VALIDATION

By

Yangdi Lyu

May 2020

Chair: Prabhat Mishra
Major: Computer Science

Hardware security validation is crucial to ensure the integrity of System-on-Chip (SoC) designs. Attackers take advantage of SoC security vulnerabilities to inject malicious functionality into the design. Validation of SoC security is challenging due to increasing SoC complexity coupled with utilization of a wide variety of third-party components from potentially untrusted suppliers. There are two major problems in SoC security validation. While there is extensive research in defining software-level vulnerabilities, there is very limited effort in classifying the potential SoC security vulnerabilities. Moreover, there is a lack of efficient techniques for simulation-based security validation as well as side-channel analysis.

In this dissertation, I propose efficient test generation techniques for SoC security validation. I have classified SoC security vulnerabilities into seven major categories, and proposed an assertion-based approach to address these vulnerabilities. To activate these security assertions, I have developed a scalable test generation framework interleaving concrete simulation and symbolic execution. I have shown that clique cover can be utilized to develop efficient tests to detect malicious implants (hardware Trojans). I have also developed an efficient test generation technique to detect hardware Trojans using current-based as well as delay-based side-channel analysis. Experimental results demonstrate that the proposed test generation techniques are effective in validation of SoC security vulnerabilities using an effective combination of simulation-based validation and side-channel analysis.

CHAPTER 1
INTRODUCTION

We are living in a connected world where a wide variety of computing and sensing components interact with each other. Reliability, safety and privacy are essential in the fabric of Internet-of-Things (IoT) as intelligent computing devices are increasingly embedded in every possible device in our daily life such as wearable devices (e.g, fitness trackers, smart watches, and medical devices), autonomous vehicles and smart homes, as shown in Figure 1-1. Many of these devices collect real-time data using sensors, and share data with other devices or the cloud without human intervention. Any failure of security and trust requirements of IoT devices may cause severe damages to critical infrastructure, endangering human life, or violating personal privacy. Since IoT devices are connecting to each other as well as communicating to the cloud to provide the required services, even one vulnerable edge device can expose the whole cluster to potential attacks. IoT applications also bring important considerations such as long application life and dynamic use-case scenarios. These IoT devices are making the world smarter, but raising the concern of security and privacy of human life.

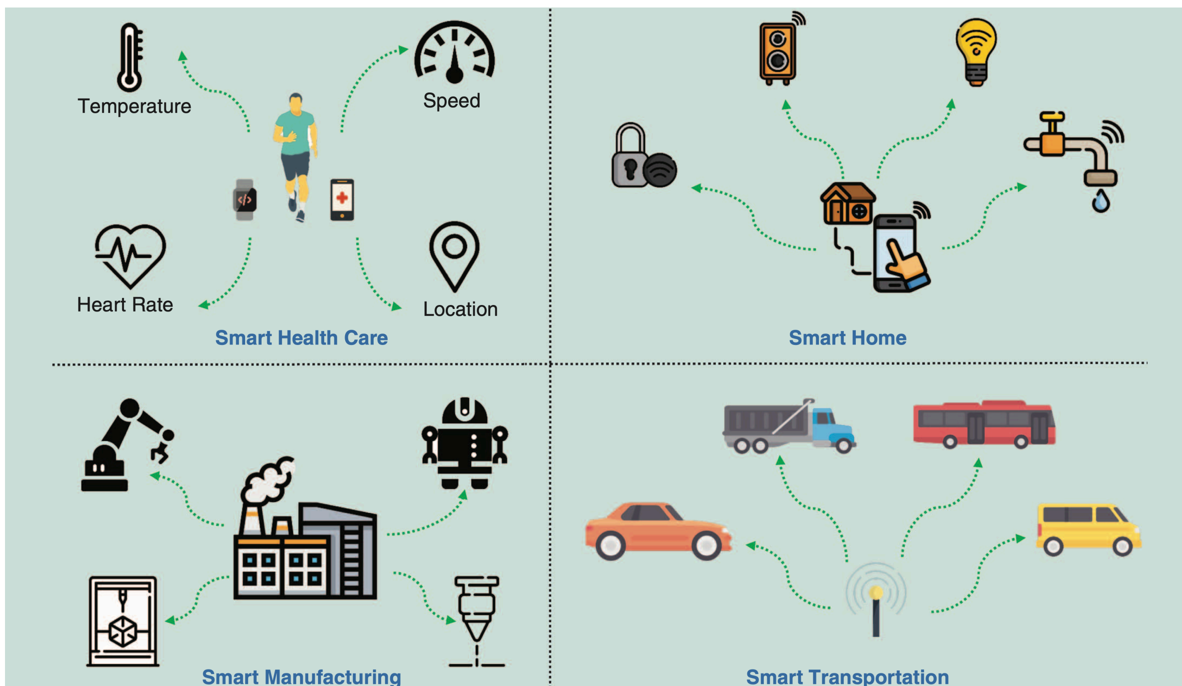


Figure 1-1. Various applications of IoT devices. [7]

System-on-Chip (SoC) is the brain behind the computing devices today. Unlike micro-controller based designs in the past, even resource constrained IoT devices nowadays incorporate one or more complex SoCs. A typical SoC consists of multiple Intellectual Property (IP) cores including processor, memory, controllers, converters, debug infrastructure, etc. As shown in Figure 1-2, a typical IoT device collects data from various types of sensors, converts the data using an analog-to-digital converter (ADC), and shares the computed results to the clouds through different forms of network protocols.

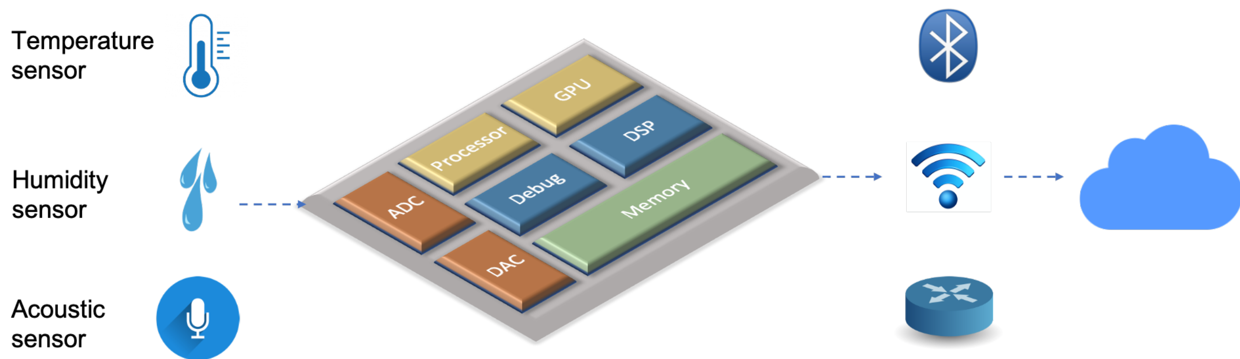


Figure 1-2. An SoC design integrates a wide variety of IPs in a chip.

The risk of attacks on IoT devices has increased more than anytime before. The attackers can explore a wide variety of vulnerabilities on these devices to mount attacks in different levels, such as hardware level, firmware level, and software level. Based on the data from MITRE/NIST Common Vulnerability Exposure (CVE) website, 43% of all vulnerabilities in 2015 were software-assisted hardware vulnerabilities [8], as shown in Figure 1-3. Among the vulnerabilities from different levels, the vulnerabilities from hardware level are the most dangerous for two reasons as shown in Figure 1-4.

1. *Significant Impact:* Hardware vulnerabilities allow attackers to mount attacks to a wide variety of IoT devices using the same hardware design. Tools such as anti-virus are not viable to protect from these attacks as these vulnerabilities are in the level of hardware. For example, Spectre [16] and Meltdown [17] explore the out-of-order execution and speculative execution in modern processors to steal information across isolated applications. These two attacks are shown to be successful in a lot of desktops, laptops, cloud servers, as well as smartphones.

2. *Difficult to Fix:* In contrast to a software vulnerability which can be modified after deployment, fixing a hardware vulnerability becomes significantly difficult and expensive. Existing approaches try to mask some of the vulnerabilities using firmware patching or utilizing in-built reconfigurable primitives. However, these approaches may not work in all scenarios.

Due to these two reasons, an attack that explores hardware vulnerabilities can be successfully repeated on every instance of IoT devices using the same vulnerable hardware.

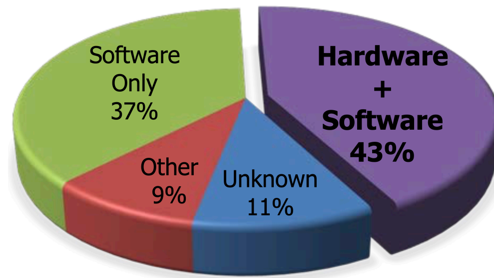


Figure 1-3. Among the recorded vulnerabilities in 2015 from MITRE/NIST CVE website, 43% were software-assisted hardware vulnerabilities. [8]

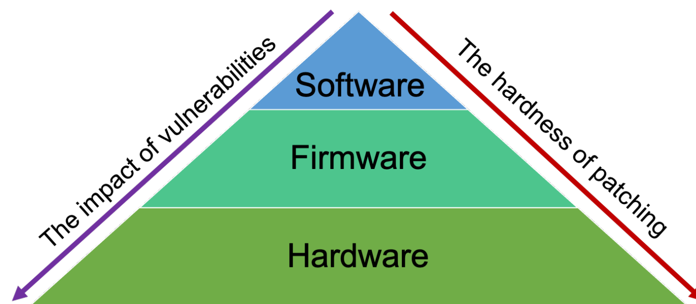


Figure 1-4. The impact of vulnerabilities and the hardness of patching increase from software, firmware to hardware.

Hardware vulnerabilities may come from three main sources. (i) As the design process contains many steps, starting from specification, synthesis, integration, to manufacturing, there are a lot of parties and tools involved. Errors and vulnerabilities can be introduced by careless designers and buggy electronic design automation (EDA) tools, as shown in the upper part of Figure 1-5. (ii) Drastic increase in SoC complexity has led to popularity in IP-based design approach. SoC design companies tend to outsource IPs to third parties to meet aggressive time-to-market constraints and reduce design cost. The vulnerabilities may be introduced by

these pre-verified hardware IPs from untrusted third-party vendors as well as the untrusted manufacturers, as shown in the lower part of Figure 1-5. (iii) The running environments of these devices are heterogeneous and possibly connected and unprotected. Vulnerabilities in one device may open a backdoor in other devices. SoC validation tries to ensure that the integrated design is free of vulnerabilities and meets all constraints including area, power, and timing overhead.

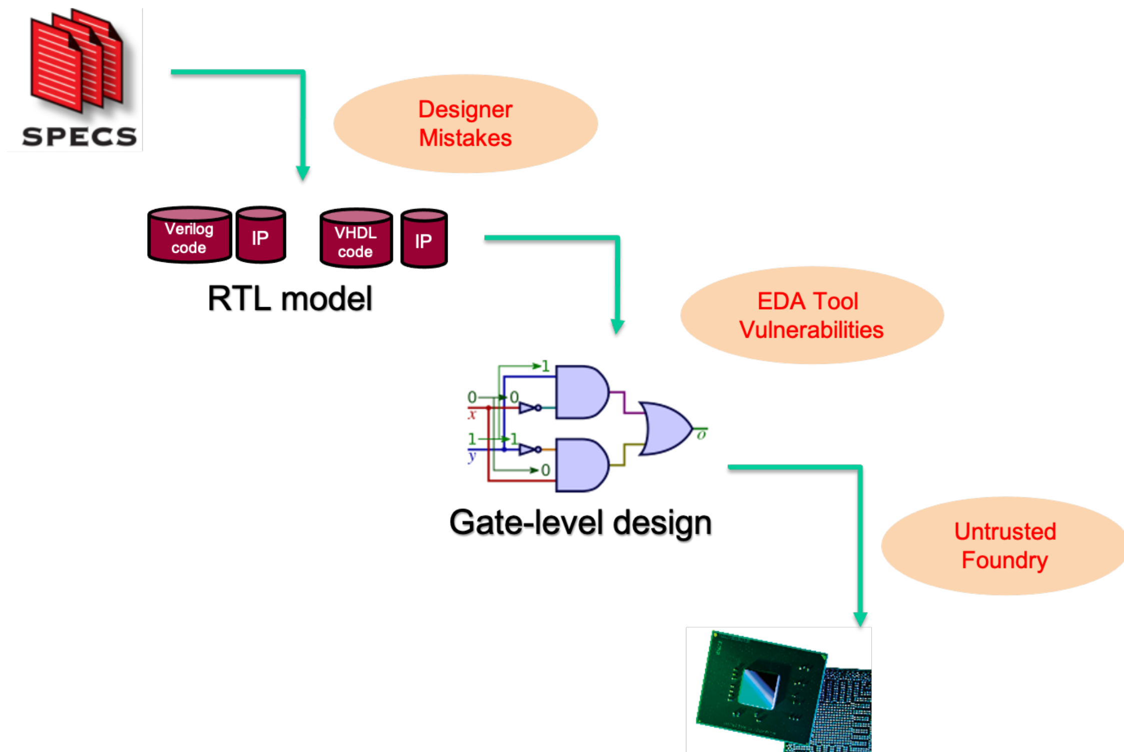


Figure 1-5. Hardware vulnerabilities come from careless designers and buggy EDA tools in hardware design, and untrusted foundries in manufacturing.

In this dissertation, I develop approaches for both simulation-based validation and side-channel analysis to detect hardware security vulnerabilities. Specifically, the focus of this dissertation is to generate high quality tests to achieve high validation coverage as well as maximize the side-channel anomaly, as shown in Figure 1-6. First, I develop a taxonomy of SoC security vulnerabilities and generate assertions for each of them. Second, I develop a test generation framework to activate all the assertions by converting assertions into equivalent branches and generating tests to activate the branches using concolic testing. Third,

I develop test generation approaches for both simulation-based validation and side-channel analysis to detect hardware Trojans. The proposed test generation framework combines advantages of both logic testing and side-channel analysis to improve the side-channel sensitivity. The rest of this chapter is organized as follows. Section 1.1 introduces existing hardware validation techniques. Section 1.2 outlines the hardware security validation challenges in SoCs. Section 1.3 summarizes the contributions of this dissertation. Finally, Section 1.4 describes the structure of this dissertation.

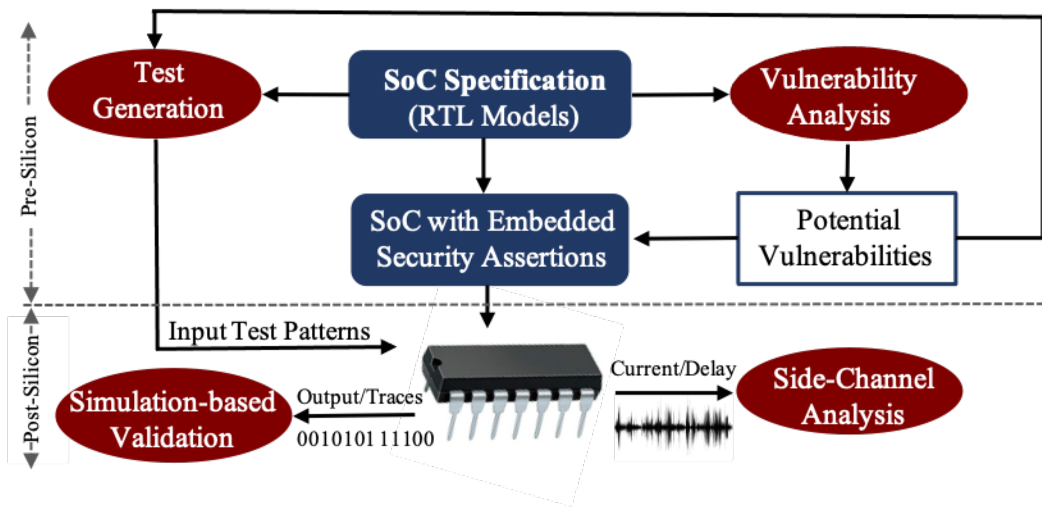


Figure 1-6. The overview of my research.

1.1 SoC Validation Methods

Validation is widely acknowledged as a major bottleneck in today's SoC design methodology. Various studies suggest that validation and verification consume about 70% of the overall SoC design effort (cost and time) [18]. There are three broad categories of SoC validation techniques: formal verification, simulation-based validation, and side-channel analysis.

1.1.1 Formal Methods

There are various types of formal methods in SoC validation, such as satisfiability (SAT) solving, property (model) checking, equivalence checking, and theorem proving [9], as shown in Figure 1-7. The goal of formal methods is to provide mathematical guarantees about the

correctness of a design. For example, property checking converts the design into mathematical representation, and formally proves the correctness of the design with respective to specific properties. A major challenge in formal methods is that it needs to covert the design into a specific formal language, which is can be error-prone. Most importantly, formal methods face the state explosion problem for large designs.

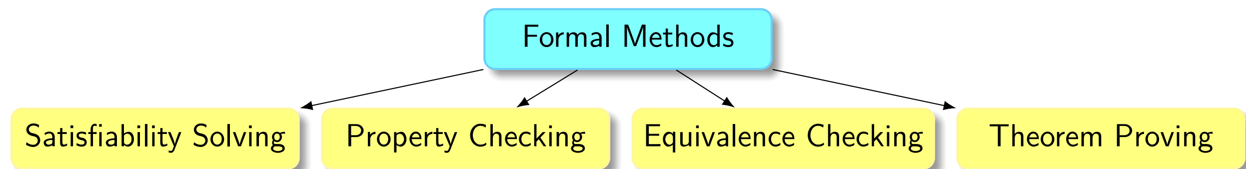


Figure 1-7. Four major categories of formal verification methods. [9]

1.1.2 Simulation-based Validation

Simulation-based validation applies tests to the design and checks the functional outputs, as shown in Figure 1-8. It is widely used for functional validation to cover various features such as all statements, all branches, etc. The quality of tests are critical to achieve high (or complete) coverage of functional scenarios. While simulation using random and constrained-random tests is scalable, it cannot guarantee full coverage of all functional scenarios. Directed test generation approaches have been proposed to generate high quality tests to achieve high coverage for simulation-based validation [19].

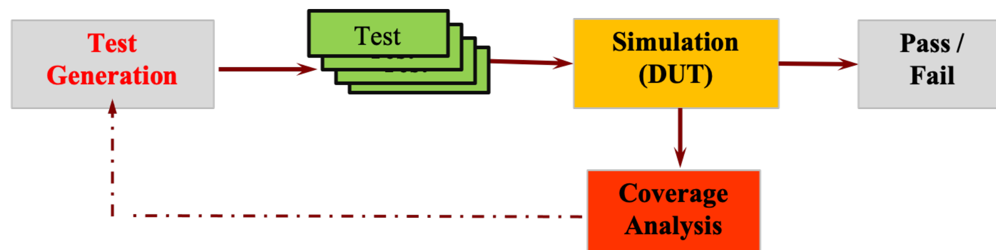


Figure 1-8. Simulation-based validation.

1.1.3 Side-channel Analysis

In contrast to simulation-based validation which relies on the functional outputs of a design, side-channel analysis compares side-channel signatures, such as power, current,

electromagnetic radiation, path delay, etc., as shown in Figure 1-9. When the side-channel difference between the golden design and the design-under-test (DUT) exceeds a threshold, the DUT is likely to be different from the golden design. The threshold exists due to the variation introduced by the manufacturing process and the noise from the measurement environment. Therefore, if the applied tests is not able to generate a large difference in side-channel signatures (higher than the threshold), side-channel analysis will fail to give any conclusion regarding the DUT.

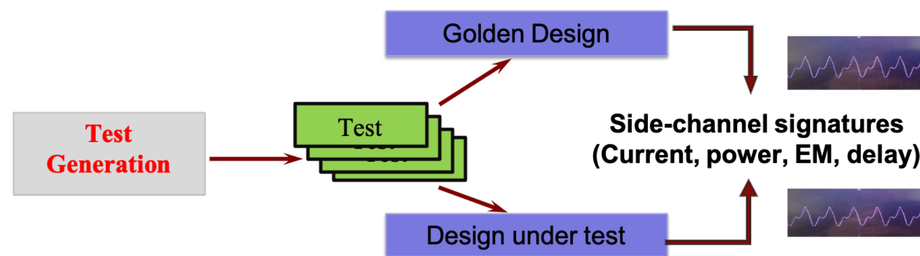


Figure 1-9. Side-channel analysis.

In this dissertation, I mainly focus on simulation-based validation and side-channel analysis. In particular, I develop novel tools and techniques to improve coverage in simulation-based validation as well as signature difference in side-channel analysis.

1.2 SoC Security Validation Challenges

There are some similarities as well as differences between SoC functional validation and SoC security validation. Some functional validation techniques can be used to detect security vulnerabilities. For example, branch coverage in functional validation can help to find hardware Trojans when they are hidden inside rare branches. Similarly, finite state machine coverage in functional validation can also help find hardware Trojans since hardware Trojans are likely to change certain states and transitions when they are triggered. However, there is one fundamental difference that introduces significant challenge to SoC security validation, i.e., the unknown nature of these vulnerabilities. Unlike functional validation where the coverage goal is well defined, security vulnerabilities are typically unknown. This section describes the major challenges in SoC security validation.

1.2.1 A Wide Variety of Vulnerabilities

While there is extensive research in defining software-level vulnerabilities, there is very limited effort in developing a comprehensive classification of potential SoC vulnerabilities. A lot of work has been done in the area of hardware Trojans, but it only represents a small fraction of the SoC vulnerability space. Specifically, it belongs to one of the many classes of vulnerabilities that I have summarized after reviewing a wide variety of security vulnerabilities listed in the National Vulnerability Database [20]. Table 1-1 shows the seven types of vulnerabilities that will be discussed in Chapter 3 in detail. As listed in Table 1-1, the vulnerabilities in SoC are so diverse that great validation efforts are needed to analyze, detect and remove all of them.

Table 1-1. Seven classes of SoC security vulnerabilities

Vulnerability	Example
Permissions and Privileges	Insufficient privilege/access checking
Resource Management	Illegal access to resources, misuse of design-for-debug infrastructures
Illegal States and Transitions	Illegal states and transitions, backdoor of undefined states/transitions
Buffer Issues	Unexpected behavior of heterogeneous buffers
Information Leakage	Information leaks from secure world to non-secure world
Numeric Exceptions	Erroneous/illegal behaviors (e.g., divide by zero)
Malicious Implants	Hardware Trojans, inserted by untrusted third party

1.2.2 Controllability and Observability

One challenge in hardware validation is the low controllability and observability of internal signals. Controllability represents the ability to control any internal signal, which requires the debug engineers to generate the correct stimulus for primary inputs. For example, it is hard to activate the trigger signal of a hardware Trojan whose trigger condition is extremely rare. Observability for validation represents the ability to reveal the internal bugs, which is also hard since the internal bugs may be hidden before it reaches the primary outputs or other observable structures, as shown in Figure 1-10. Even with the help of Design-for-Debug (DfD) architecture, the visibility is very limited for internal signals (e.g., 256 signals can be traced

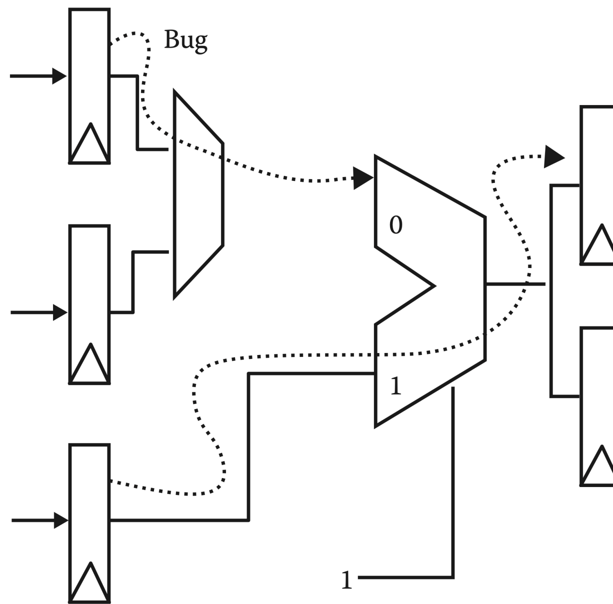


Figure 1-10. Poor observability in hardware validation. [10]

for 2048 cycles in a multi-million gate design) during post-silicon execution. Assertion-based validation is a common practice in industry for functional validation. The embedded assertions can catch any unexpected behavior which increases the observability of internal activities inside the design. For example, an assertion can check that the output of an adder is equal to the sum of its inputs whose implementation may be different in different designs. Any bug in the design that violates the predefined properties in assertions can be easily detected. The observability of internal states enables faster localization of errors, which reduces the overall validation time significantly. Although assertions have been successful in functional validation, it is not clear if non-functional behaviors (e.g., security) can be posed as constraints (assertions). For example, security validation needs to check the vulnerability of potential information leakage. However, there are no easy constraints to ensure that there is no path to leak information from secure world to unsecure world.

1.2.3 Lack of Effective and Scalable Validation Techniques

Similar to functional validation, security vulnerabilities can also be detected using simulation-based approaches and side-channel analysis. Simulation using random and constrained-random tests is widely used in traditional validation methodology. Unfortunately,

even billions or trillions of constrained-random tests cannot cover many complex and corner-case scenarios in today's industrial designs. Another problem in random and constrained-random tests is the slow speed of simulation. With billions or trillions of tests, it is expected to take months to finish simulation. Directed tests are promising as they can achieve comparable coverage with significantly less tests. Directed testing is also promising in covering specific cases that are not covered by random or constrained-random tests. In contrast to simulation-based approaches which focus on the functionality of the design, side-channel analysis checks if a vulnerability exists by comparing the side-channel signatures from the golden design and the design-under-test. However, the sensitivity of side-channel analysis achieved by state-of-the-art approaches is too small to detect vulnerabilities under current process variations and environmental noise.

There are promising formal approaches to solve specific problems related to hardware security vulnerabilities, such as equivalence checking in arithmetic circuit to detect hardware Trojans [9], and bounded model checking for activating security assertions. However, these approaches are not effective and scalable to detect a wide variety of vulnerabilities in SoC security validation due to three fundamental challenges: increasing complexity of finite state machines, stealthy nature of security vulnerabilities, and exponential validation space. The remainder of this section describes these challenges in detail.

Complex Finite State Machines: For the ease of illustration, I use cache coherence protocol as an example to explain why complex finite state machine is a challenge for SoC security validation. System designers incorporate multi-core processors to meet the increasing performance requirements. To address the memory bottleneck, caching has been the most effective approach to reduce the memory access time for several decades. When the same data is cached by different processors, cache coherence protocols are employed to guarantee that a read always returns most recently written data. This rule requires the whole system to monitor every copy of the same data across multiple processors. To boost the overall performance and reduce the number of flushing and loading, the design of cache coherence protocols is

also becoming more and more complex, from the simplest MSI to MESI and MOSI, then to MOESI [21] and MESIF [22]. With the increasing number of cores and complexity of cache coherence protocols, the compound state and transition space is getting exponentially complex. The first complexity comes from the exponentially growing number of states and transitions. For example, in 8-core and 16-core MESI protocols, there are 5 thousands and 2.6 million transitions, respectively. It is infeasible to generate tests and traverse every state with more cores. The second complexity comes from the obscure structures. Although the FSM of each cache controller is easy to understand, the product finite state machine for modern cache coherence protocols is quite obscure. For example, Figure 1-11 shows the obscure finite state machine of a MESI protocol with only three cores. With more cores, the structure of the FSM is expected to be too obscure to apply any analysis.

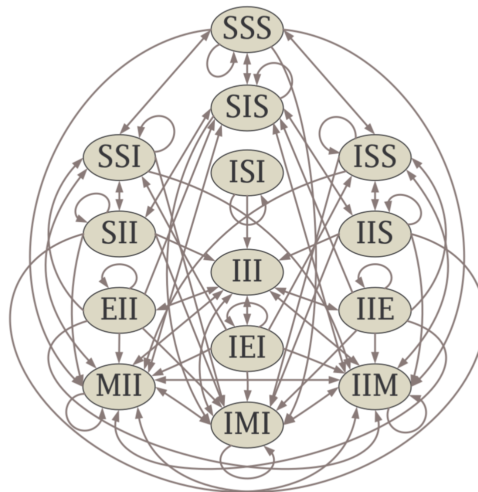


Figure 1-11. Full system FSM of three processor MESI-based system. [11]

Stealthy Nature of Vulnerabilities: To illustrate the stealthy nature of vulnerabilities, I use malicious implants as an example. Hardware Trojans [23, 24] are malicious modifications incorporated during the System-on-Chip (SoC) design cycle [23, 25–28]. As IC design and fabrication process becomes more and more globalized, the threat of hardware Trojan attacks is increasing due to potential malicious modifications at different stages of the design and fabrication process [25, 29]. A Trojan normally consists of a rare trigger condition and a

payload, as shown in Figure 7-1. Trigger condition is carefully crafted such that it is only activated under extremely rare conditions, and the functionality of a design remains exactly the same as the golden design under most of the running cases. The inherent stealth of trigger condition makes hardware Trojan detection a challenging problem. The number of choices to construct trigger conditions grow exponentially with the complexity of SoCs. For example, even for a small ISCAS benchmark (c880 with only 451 gates) [30], there are approximately 10^{11} triggers possible with only four trigger points. The number would be exponentially higher if we consider triggers with different number of trigger points. Clearly, it is infeasible to generate and apply so many directed tests to activate Trojan triggers even for a tiny benchmark.

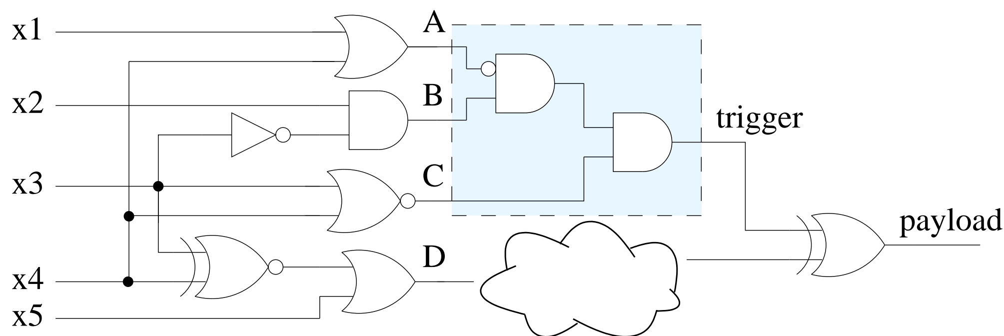


Figure 1-12. An example hardware Trojan.

Exponential Validation Space: As the complexity of SoC designs grows exponentially, the validation efforts also grow exponentially to achieve reasonable coverage analysis. Millions of random or constrained-random tests are able to quickly cover majority of functional scenarios (targets). However, it is not always possible to cover all scenarios using these tests and the number of remaining targets can be huge (hundreds or thousands) in case of today's industrial designs. For specific targets, random simulation cannot guarantee to detect them within debug budgets. For example, Figure 1-13 shows the scenario that three random simulation paths can cover the target T_1 but cannot cover the target T_2. Verification engineers usually manually write test cases to cover the remaining hard-to-activate scenarios. Due to the increasing design complexity, the number of remaining hard-to-activate scenarios can be exponential. While such manual test case development is possible for small designs, it would be

infeasible to develop directed test for large designs. Coming up with manual test cases can be both error-prone and time-consuming due to many trial-and-error iterations.

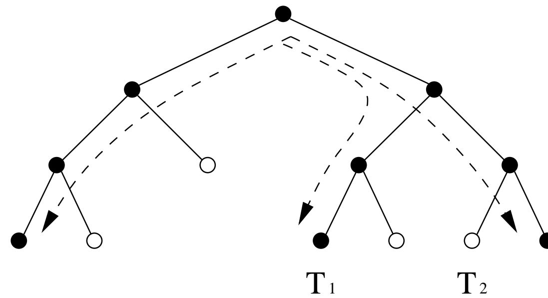


Figure 1-13. Random simulation cannot guarantee the coverage of all targets.

Automated test generation is necessary to achieve full coverage of all remaining targets that are not covered by random simulation. There has been extensive research in test generation for code coverage in RTL models using formal methods [31, 32]. Although formal methods can cover specific scenarios directly, they require tedious and error-prone translation from real design to their own languages and suffer from state explosion for large designs. Semi-formal approach, such as concolic testing, is promising to address the scalability problem by combining the advantages of random simulation and formal methods to activate targets efficiently. The idea of concolic testing comes from software testing domain [33, 34]. It uses depth-first-search (DFS) or breadth-first-search (BFS) to quickly find input patterns to activate specific bugs in software. However, none of these search strategies can work for large programs due to path explosion problem.

1.3 Research Contributions

Directed test generation is a promising and efficient method for hardware security validation. Compared to random simulations, directed test generation can reach a specific goal with a drastically small number of tests. This dissertation mainly focuses on directed test generation for validation of hardware security. Specifically, it makes the following fundamental contributions: (1) the taxonomy of SoC security vulnerabilities, (2) simulation-based validation by exploring the specification, structures and functionality, and (3) side-channel analysis based

on dynamic current and path delay to detect malicious implants (hardware Trojans). The reminder of this section outlines these contributions.

Taxonomy of Security Vulnerabilities: Existing SoC validation techniques mainly focus on the functional behaviors defined in the specification. Traditionally, SoC security vulnerabilities are not considered as expected functional behaviors. For example, an unspecified transition in finite state machine (FSM) is one of the main sources of security vulnerabilities. While assertion-based validation (ABV) is the de facto standard for functional validation, there are no prior efforts to define and monitor SoC security vulnerabilities. Given the importance of SoC security, I classify security vulnerabilities into seven categories, and propose a framework for defining and utilizing SoC security assertions to detect each category of vulnerabilities.

Code Coverage for Multiple Targets: Concolic testing is promising to cover the rare scenarios that are not covered by millions of random tests. It addresses the state explosion problem in formal methods. However, existing concolic testing approaches in RTL model are not effective in path selection and not scalable due to overlapping search. To address these two problems, an efficient technique for multi-target test generation using concolic testing is proposed. It fully utilizes information from the previous search. Specifically, this approach improves the overall performance by the following optimizations: (i) efficient pruning of targets that can be covered by the tests generated for activating other targets, (ii) clustering of related targets to drastically reduce the test generation time, where targets in the same cluster usually share a common simulated path, and (iii) a novel edge realignment technique to effectively evaluate the distance between a simulated path and a target.

Assertion Coverage by Concolic Testing: While existing test generation using model checking is promising in activating assertions, it cannot generate directed tests for large designs due to the state space explosion problem. I propose an automated and scalable mechanism to generate directed tests using a combination of symbolic execution and concrete simulation of RTL models. The proposed methodology consists of two major steps. The first step converts these assertions to branch statements and embeds them into the design. Then, it utilizes

concolic testing to generate a compact test set to efficiently cover (activate) the target branches (assertions). The generated test vectors are guaranteed to activate the corresponding assertions non-vacuously. Compared to the exponentially growing memory requirements in model checking, the memory requirement grows linearly in the proposed approach.

Scalable Cache Coherence Protocol Validation: To address the scalability concerns in validating FSM of cache coherence protocols with many cores, my research combines on-the-fly test generation technique [35] with a quotient space based approach. This approach first analyzes the state space structure of their corresponding global FSMs and decomposes them into several components with simple structures. Then, it utilizes the symmetric structure of protocol state space to efficiently cover all states and transitions. Next, quotient space based scalable test generation algorithms trade-off between functional coverage and verification effort to cover important state transitions within limited verification budget. Quotient space guarantees selection of important transitions by utilizing equivalence classes, and omits only similar transitions to provide scalable test generation framework. The experimental results demonstrated the effectiveness on systems with many cores and complex cache coherence protocols, making it suitable for future multi-core architectures.

Trigger Coverage using Clique Cover: Trigger activation is a major challenge due to the exponentially large space that an adversary can exploit to construct trigger conditions. Conventional validation approaches using millions of random test vectors or ATPG test vectors are not effective in activating extremely rare and unknown trigger conditions. To address the fundamental challenge of activating rare triggers, I propose a new test generation paradigm for Trigger Activation by Repeated Maximal Clique sampling (TARMAC). The basic idea is to utilize a satisfiability modulo theories (SMT) solver to construct a test corresponding to each maximal clique. It makes three fundamental contributions: (1) it proves that the trigger activation problem can be mapped to clique cover problem, and the test vectors generated by covering maximal cliques are complete and compact, (2) it proposes efficient test generation algorithms to activate trigger conditions by repeated maximal clique sampling, and (3) it

outlines an efficient mechanism to run the clique sampling in parallel to significantly improve the scalability of our test generation framework.

Current-based Side-Channel Analysis: I propose an efficient test generation technique to facilitate side-channel analysis utilizing dynamic current. My proposed approach effectively searches for efficient tests that can drastically improve the side-channel sensitivity - making Trojan detection feasible in practice. To increase the overall sensitivity, my approach exploits the input affinity to identify test patterns that can maximize switching in the suspicious (target) region while minimize switching in the rest of the circuit in order to significantly improve the side-channel sensitivity. This approach shows significant improvement in sensitivity to detect the majority of Trojans, while the state-of-the-art approaches can detect less than 1% Trojans.

Delay-based Side-Channel Analysis: Existing delay-based side-channel analysis techniques have two major bottlenecks: (i) they are not suitable in detecting Trojans since the delay difference between the golden design and a Trojan inserted design is negligible, and (ii) they are not effective in creating robust delay signatures due to reliance on random and ATPG based test patterns. In this dissertation, I propose an efficient test generation technique to detect Trojans using delay-based side channel analysis. First, I design a lightweight and effective logic testing algorithm to generate tests for delay-based side-channel analysis. The generated tests assume no preliminary information about critical paths or trigger conditions. Next, I perform a Hamming-distance based reordering of the generated tests. The reordering is based on a distance evaluation method that can increase the probability of constructing a critical path from the trigger to the payload.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 surveys existing security validation approaches. In Chapter 3, a wide variety of common SoC security vulnerabilities are identified and their corresponding classes of assertions are proposed. Chapter 4 describes an automated test generation technique for activating multiple targets in RTL models using

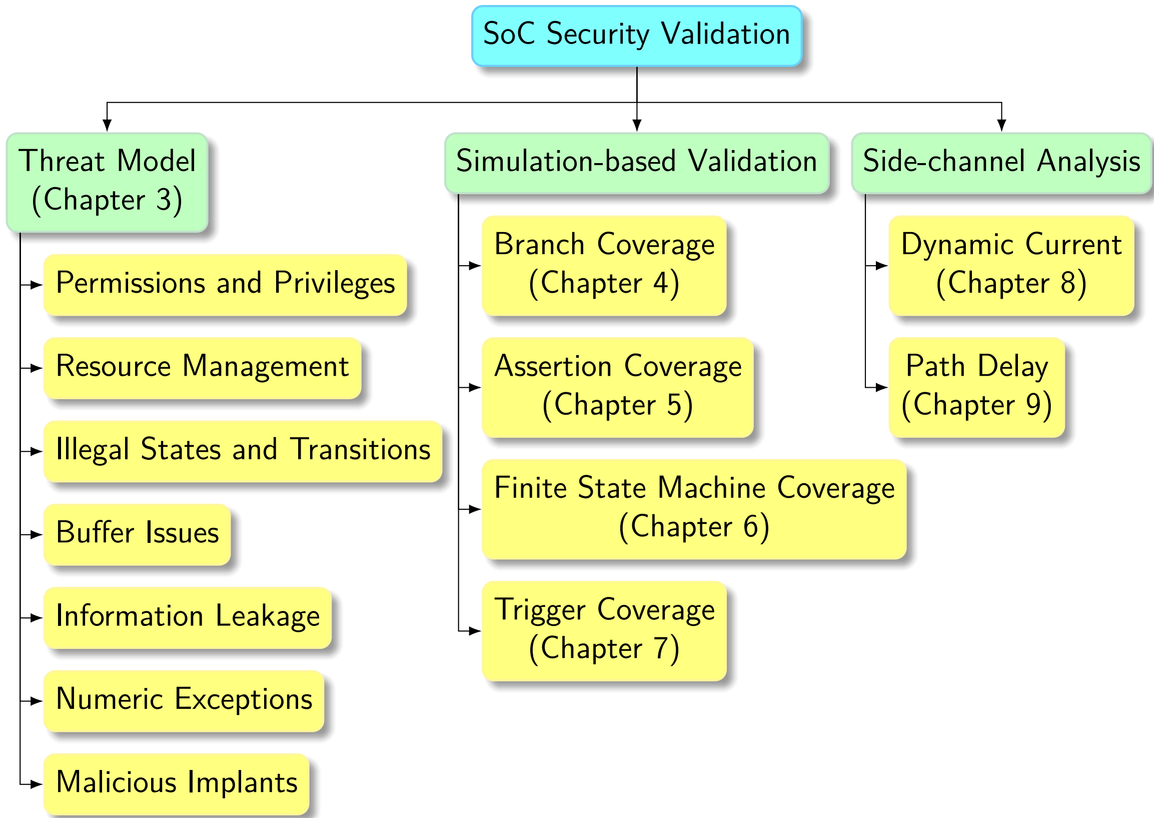


Figure 1-14. Dissertation outline.

concolic testing. Chapter 5 shows how concolic testing can be applied to RTL models to activate assertions. Chapter 6 presents a scalable on-the-fly test generation technique to validate the FSM of cache coherence protocols. Chapter 7 demonstrates an effective test generation approach to activate trigger conditions by mapping it to the problem of clique covering. Chapter 8 and Chapter 9 describe an efficient test generation technique for detecting hardware Trojans using dynamic current and path delay, respectively. Finally, Chapter 10 concludes this dissertation.

CHAPTER 2
BACKGROUND AND RELATED WORK

There are extensive research efforts for hardware security validation. These approaches can be broadly classified into three categories: formal methods, simulation-base validation, and side-channel analysis. This chapter reviews these existing approaches that are related to this dissertation.

2.1 SoC Security Validation using Formal Methods

Instead of relying on the quality of generated tests to reveal the vulnerabilities in a design, formal methods utilizes mathematical models to verify that the design (implementation) satisfies its specification. There are four widely used forms of formal verification methods.

- **Satisfiability solving** determines if a given Boolean formula can be true. There are many validation problems that can be converted into satisfiability problem. For example, to validate if an implementation is equivalent to its specification, a Boolean formula can be constructed by computing (XOR) of the functional outputs of these two designs. If the SAT solver returns assignments to the variables that make the formula to be true, the implementation and the specification are different.

- **Equivalence checking** [9, 36, 37] verifies if two designs are equivalent or not. There are two common types of techniques for equivalence checking. The first type utilizes model checking to verify the equivalence using Binary Decision Diagrams (BDDs). The other type utilizes SAT solvers to return a counter-example if the two designs are different by converting the designs to Conjunctive Normal Form (CNF).

- **Property checking** [38] verifies if a design satisfies a set of properties. The design and properties are first converted to formal languages. Both the design and the properties are fed into a model checker. The model checker either finds a counter-example or proves that the property holds in the design. Property checking can be implemented using BDDs as well as SAT-based bounded model checking.

- **Theorem proving** [39] builds mathematical formulas to represent the behavior of a design (implementation) and evaluate these formulas against a specific requirement (security specification). It needs to formulate a theorem to describe the requirement. Next, it needs to prove the theorem using a set of axioms and facts that can be derived from the specification.

2.2 SoC Security Validation using Simulation-based Validation

Simulation-based validation aims at developing tests to simulate the design and achieve high coverage goals. There are different methods to generate tests, including

random/constrained-random approaches, directed testing generation using formal methods, statistical methods, and concolic testing which interleaves simulation and formal methods.

2.2.1 Random/Constrained-Random Simulation

Simulation using random and constrained-random tests is widely used in both industry and academia due to its good scalability. For example, random/constrained-random approaches are widely used to validate the states and transitions in FSM. Wood et al. [40] used random tests to verify the memory subsystem of SPUR machine. Genesys Pro test generator [41] from IBM extended this direction with complex and sophisticated test templates. To reduce the search space, Abts et al. [42] introduced space pruning technique during their verification of the Cray processor. Wagner et al. [11] designed the MCjammer tool which can get higher state coverage than normal constrained random tests. Since an uncovered transition can only be visited by taking a unique action at a particular state, it may not be feasible for a random test generator to eventually cover all possible states and transitions. To address this problem, some random testers are equipped with a small amount of memory, so that the future search can be guided to the uncovered regions. Unfortunately, unless the memory is large enough to hold the entire state space, it is hard to achieve full coverage by such guided random testing.

2.2.2 Directed Test Generation using Formal Methods

In spite of the fast test generation time in random and constrained-random approaches, their coverage/performance is poor. Validation using billions or trillions of random tests cannot provide 100% coverage. The uncovered scenarios will likely be hard-to-detect (rare) branches and corner-case scenarios in today's industrial designs. Verification engineers usually write specific (directed) test cases manually to cover the remaining hard-to-activate scenarios such as corner cases and rare events. While manual test development is possible for small designs, it would be infeasible to develop directed tests manually for complex SoC designs. Moreover, manual development of test cases can be both error-prone and time-consuming due to many trial-and-error iterations in complex designs. Automated test generation is necessary to overcome these issues.

Formal methods, such as model checking [4, 35, 43–45], is effective in automated generation of directed tests for property checking. To activate a specific target (functional behavior), the negated version of a property (functional behavior) is fed into a model checker, which will return a counterexample as the test that can activate the target. Binary Decision Diagrams (BDD) based BMC [46] and SAT-based BMC [38] are two widely used model checking methods [47]. From non-deterministic finite automata (NFA), Tong et al. [44] utilized model checking to generate test for assertions with the assumption that the signals in assertions refer to the primary inputs. This type of approaches have a few bottlenecks. In order to enable test generation, they restrict the assertions to have variables of only specific types (e.g., primary inputs). Model checking is also used to validate the correctness of a cache coherence protocol. It can prove mathematically whether the description of certain protocol violates a required property [48].

There are two major problems in applying formal methods directly to the design. First, formal methods expect formal specification and require translation from specifications or Hardware Description Language (HDL) models to their supported formats. The extra procedure of conversion to formal specification may introduce errors. Second, the complexity of real world designs usually exceeds the capacity of the model checking tools, leading to state space explosion. Extensive research has been devoted to reduce the model checking complexity during test generation using various design/property decomposition as well as learning techniques [35, 49, 50]. In spite of these extensive efforts, it is infeasible to generate directed tests using model checking based approaches due to inherent state explosion problem while dealing with complex properties as well as large designs.

2.2.3 Statistical Methods

Logic testing [12, 51–54] is the main approach to detect hardware Trojans by comparing the outputs of an implementation to a golden specification. ATPG based logic testing is an effective method to find tests of a design using fault simulation. With the help of Design-for-Testability (DfT) structures, such as scan chain, ATPG treats all sequential logic

gates as combinational logic gates under full scan mode and is able to generate test efficiently. Logic testing needs to fully activate the Trojan and propagate its effects to observable outputs. Given the exponential test space complexity, directed testing may not be feasible to directly activate a rare trigger condition and propagate the Trojan effects to observable outputs. Statistical test generation techniques (e.g., N-detect approach [12]) are promising for unknown Trojans. The basic idea is to activate all rare signals as much as possible (one or more at a time) to increase the likelihood of activating the actual (unknown) trigger consisting of rare signals. Extensive research has been done on statistical test generation combining ATPG and N-detect paradigm [12, 55]. The authors observed that if the generated test patterns are able to satisfy all rare values N times (N-detect criterion), it is highly likely that the unknown rare trigger conditions are satisfied when N is sufficiently large.

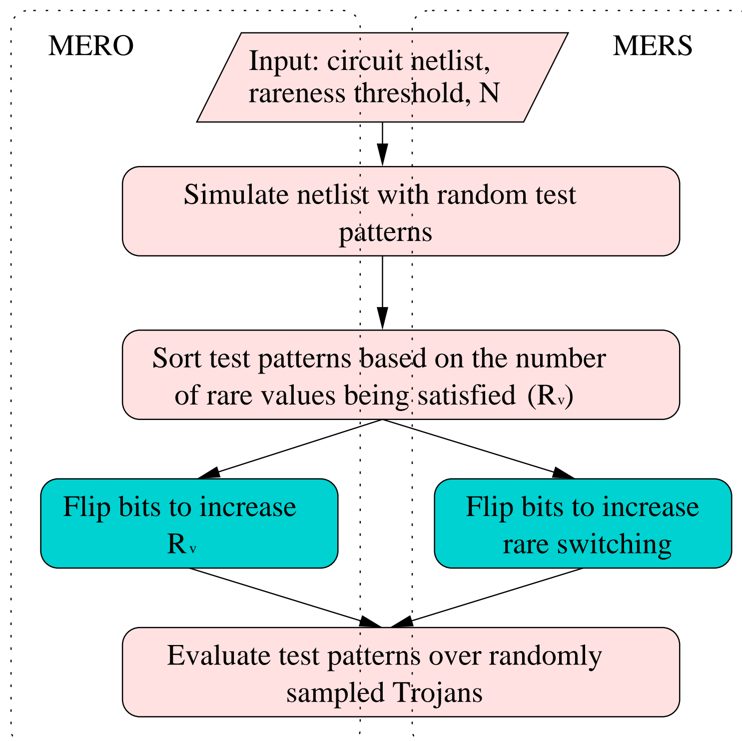


Figure 2-1. The frameworks of MERO [12] and MERS [6].

In [12], the authors proposed a tool named MERO to generate N-detect test for logic testing. The framework of MERO (N-detect) [12] is shown in the left part of Figure 2-1.

MERO is a constrained-random approach to achieve the N-detect criterion through flipping bits. It starts from a large number of random test patterns. Then, it simulates the netlist with each random test pattern and gets the number of rare values being satisfied (R_v). Next, these patterns are sorted based on R_v. The profitable test patterns (with greater R_vs) are visited earlier than the ones with smaller R_vs. For each random test pattern, each bit is flipped one after the other to improve its quality. If the flipping of some bit can improve the N-detect criterion, the flipping is accepted. Otherwise, the flipping is reversed. The modified test pattern is put into the final test set if it is helpful in improving the N-detect criterion. MERO is shown to be effective in small designs (e.g., ISCAS benchmarks [30, 56]) with relatively easy-to-activate trigger conditions (with four rare signals, and larger than 0.1 rareness threshold). However, MERO is unsuitable for large designs (scalability problem) as well as hard-to-detect triggers [57]. Another problem in N-detect approach is the long test generation time, since it needs one simulation for each bit flipping. It is important to note that N-detect criterion is a globally evaluated requirement, which needs each rare signal to be activated by at least N times. Therefore, the checking of improvement on N-detect criterion needs to consider all the generated test patterns together. This global evaluation method prevents N-detect approach from running in parallel.

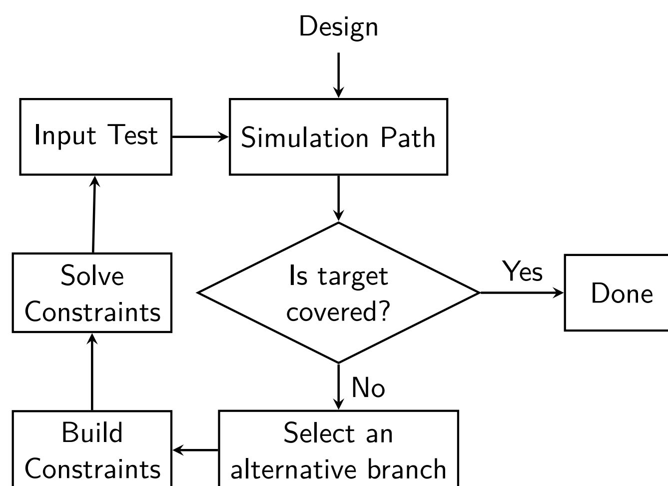


Figure 2-2. The main steps of concolic testing.

2.2.4 Concolic Testing

Concolic testing is a promising semi-formal test generation technique by interleaving concrete simulation and symbolic execution. Unlike formal method based approaches that explore all possible (exponential) execution paths at the same time (and lead to state space explosion), concolic testing explores only one execution path at a time and reaches a target statement by alternating one branch. The major steps of path exploration are simulate, generate constraints, select a new path and solve constraints, as shown in Figure 2-2. After obtaining a concrete path from simulation, concolic testing executes the given design symbolically by updating variables with assignments and constraints obtained from the concrete path. To explore new paths, an alternative branch is selected from previous path and symbolically solved by constraint solvers. If the new constraints are satisfiable, an input vector will be returned and used to generate a new simulated path. If the simulated path covers the target, the input vector is added to the test set. As concolic testing examines one path at a time, it addresses state explosion problem associated with test generation using formal methods [19]. However, the performance of concolic testing is decided by how the new path is selected, or ***alternate branch selection*** (Figure 2-3). When profitable branches are selected, the simulated path will quickly reach the target. On the other hand, selecting wrong branches may lead to longer test generation time, or even failure to activate the target. Another factor that affects concolic testing is the ***initial path*** that we choose to start concolic testing, which is usually a random test or a manually developed test. When the initial path is already closer to the target, it is easier to reach the target and less likely to be lost in the large search space.

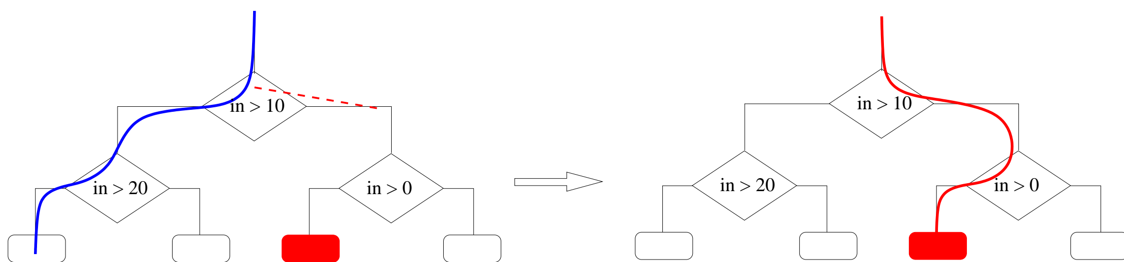


Figure 2-3. Alternative branch selection in concolic testing.

2.2.4.1 Concolic Testing of Software Designs

There are extensive research efforts in applying concolic testing on software designs to cover functional events [33, 34, 58]. Each approach utilizes different path selection heuristics and optimizations to achieve specific coverage goals. To quickly cover targets, structural information from control flow graph (CFG) is analyzed to guide path exploration [59, 60]. Another semi-formal method that is similar to concolic testing in generating test for a specific target is called symmetric backward execution [61–63], which explores paths from a target back to an entry point. Instead of trying to improve overall coverage, such as KLEE [58], its goal is typically to satisfy all preconditions to execute a specific statement. While these approaches are successful in software domain, they are not directly applicable on hardware (SoC RTL models) designs since they have to deal with unrolling multiple CFGs with complicated communication between different models and different clock domains.

2.2.4.2 Concolic Testing of Hardware Designs

Concolic testing has been shown effective in hardware/software co-validation on virtual platforms [64–67] and high-level modeling using SystemC [68]. While there are some early efforts on applying concolic testing to RTL models [69, 70], they are applicable on simple designs with restricted Hardware Description Language (HDL) features. There are some recent efforts in applying concolic testing on RTL models. Approaches based on uniform test generation techniques utilize various search techniques, such as depth-first search (DFS), breadth-first search (BFS), etc., to cover as many branches as possible. Ahmed *et al.* [2] proposed QUEBS to balance exhaustive and restrictive search techniques by limiting the number of times a branch can be selected. While uniform test generation is promising, it suffers from the exponentially growing number of paths which makes exhaustive searching impractical (path explosion problem) for covering a number of selected targets. As a result, is it not suitable for large designs. Another promising approach utilizes static analysis of CFGs to guide searching of alternate branches [3]. As demonstrated in Chapter 4, it leads to exploration of undesired paths in many scenarios, and as a result, it still faces path explosion problem.

I propose an efficient path exploration scheme to improve the quality of explored paths to address the path explosion problem in concolic testing. Moreover, I propose clustering and learning techniques to minimize wasted efforts in overlapping searches while generating tests to activate multiple targets.

2.3 SoC Security Validation using Side-channel Analysis

Side-channel analysis has been widely used for hardware Trojan detection [6, 13, 71–79]. It examines side effects of the inserted Trojans, such as power, dynamic current, and path delay. Compared to functional validation for hardware Trojan detection, side-channel analysis does not require the Trojan to be fully activated or to propagate its effect to the observable outputs. However, it faces the challenge from the process variation and environmental noise. As transistor dimensions continue to shrink, it introduces increasing process variations across integrated circuits (ICs) of the same design. Since the sizes of hardware Trojans are small (e.g., few gates in a million-gate IC), the deviation introduced by the Trojans is typically negligible with respect to process variation and environmental noise. A measured small deviation in side-channel signature cannot conclude the existence of a Trojan. The challenge is how to automatically generate high quality test patterns that can maximize the anomalies in side-channel signatures, e.g., maximizing switching difference for current-based analysis, or sensitizing critical paths for delay-based analysis. In this section, I will introduce existing approaches based on two commonly used side-channel signatures in hardware Trojan detection.

2.3.1 Dynamic Current based Side-Channel Analysis

Dynamic current based side-channel analysis measures transient current both in the golden design and the design under test, as shown in Figure 2-4. If the measured signals from these two designs vary by a threshold, a Trojan is suspected to be present. Huang et al. [6, 80] tried to combine the advantages of logic testing and side-channel analysis by extending the idea of N-detect test for side-channel analysis. They proposed a test generation framework called MERS to maximize the sensitivity of dynamic current. The frameworks of MERS and MERO are similar, as shown in Figure 2-1. MERS generates compact test patterns to let each

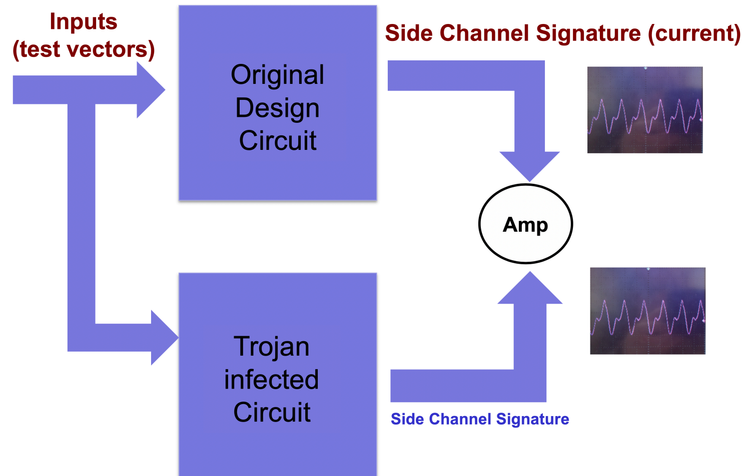


Figure 2-4. Side-channel analysis detects hardware Trojans by comparing the difference of side channel signatures.

rare node switch from its non-rare value to its rare value N times, increasing the probability of partially or fully activating a Trojan. After generating MERS test patterns, the authors proposed simulation based reordering (called MERS-s) to decide the order of applying test patterns to maximize side-channel sensitivity. Although MERS improves the sensitivity by 1033% over random test patterns [6], the increase in side-channel sensitivity is marginal, which is less than 1% in the majority benchmarks [6] whereas process variations can be 7-17% [81] in today's CMOS circuits. The low side-channel sensitivity in [6] is due to the inherent restriction of reordering within the set of test patterns generated by MERS. The other disadvantage of MERS is that it inherits the same bad performance of the N-detect approach and makes the test generation time even longer by reordering tests. The reordering step of MERS in maximizing side-channel sensitivity requires a large number of simulations and is not able to run in parallel (the same problem as the N-detect approach). Chapter 8 proposes an approach to effectively search for efficient tests that can drastically improve the side-channel sensitivity, and allow the searching process run in parallel - making Trojan detection feasible for large designs in practice.

2.3.2 Path Delay based Side-Channel Analysis

Path delay based side-channel analysis is beneficial compared to other side-channel parameters as the delay of each output can be measured independently, and an inserted Trojan may affect multiple observable outputs. The delay is expected to be greater than the delay in the golden design with extra gates inserted.

A hardware Trojan has two types of effects over path delay. The first one comes from the change of fan-out. In Figure 2-5A, as the trigger points connect to an extra gate compared to the golden design, the gates that produce these signals will change their capacitive load. As a result, the propagation delay of these gates will change. The other type of impact is from the extra gates that are inserted by the payload. For example, the XOR gate in Figure 2-5B is inserted to change the value of the original signal when the trigger is activated. This extra XOR gate adds to the total path delay of any path passing through it. The delay anomalies introduced by the extra gates are typically larger than delay anomalies introduced by capacitive loading effects of fan-out HT gates [13]. As path delays cannot be measured directly, most existing research use static timing analysis tools to measure delay. One common approach is to generate Standard Delay Format (SDF) file that contains information of each gate and connection in the design, and then utilize gate-level simulation to compute the delays. Process variations can be added either in the SDF file or in the simulation phase.

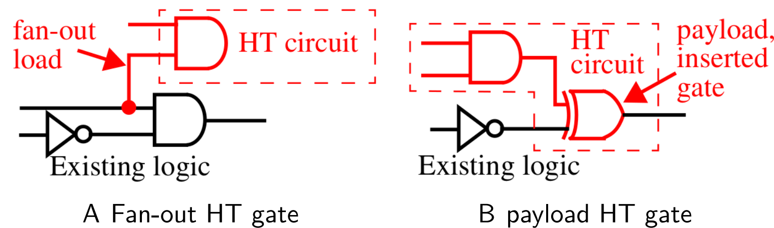


Figure 2-5. Two types of impact on path delay from hardware Trojans [13].

The main challenge in path delay based approach is to find suitable input patterns (tests) that can reveal delay difference introduced by the Trojan. Existing approaches apply a static analysis on the design to find all possible paths, and use Automatic Test Pattern Generation

(ATPG) tools to generate test patterns that are able to sensitize these paths. For example, Jin et al. [72] used Synopsys TetraMAX to analyze the design and generate test patterns to cover every path. However, this approach is time-consuming and not scalable for large designs since the number of possible paths grows exponentially with the size of the design. In addition, the small delay difference introduced by hardware Trojans is likely to be dominated by large process variation and environmental noise. In Chapter 9, I propose an approach to significantly increase the delay difference by changing the critical paths to offset possible noise.

2.4 Summary

This chapter surveyed a wide variety of SoC security validation techniques. Specifically, it provided a comprehensive review of existing SoC validation approaches using formal methods, simulation-base validation as well as side-channel analysis.

CHAPTER 3
SYSTEM-ON-CHIP SECURITY ASSERTIONS

Existing SoC validation techniques mainly focus on the functional behaviors defined in the specification. Traditionally, SoC security vulnerabilities are not considered as expected functional behaviors. For example, an unspecified transition in finite state machine (FSM) is one of the main sources of security vulnerabilities. Assertions are widely used for functional validation as well as coverage analysis for both software and hardware designs. Assertions enable runtime error detection as well as faster localization of errors. While there is a vast literature on both software and hardware assertions for functional validation, there are no prior efforts to define and utilize assertions to monitor System-on-Chip (SoC) security vulnerabilities. Given the importance of SoC security, in this chapter, we identify common SoC security vulnerabilities by analyzing the design. To monitor these vulnerabilities, we define several classes of assertions to enable runtime checking of security vulnerabilities. Our experimental results demonstrate that the security assertions generated by our proposed approach can detect all the inserted vulnerabilities while the functional assertions generated by state-of-the-art assertion generation techniques fail to detect most of them.

The framework for defining and utilizing SoC security assertions is shown in Figure 3-1. The framework consists of two main steps. First, it performs vulnerability analysis on a given SoC design and identifies potential vulnerabilities. Next, security assertions are generated from the vulnerabilities and inserted into the SoC design. The purpose of this chapter is not to provide a laundry list of vulnerabilities for SoC design. Instead, this chapter identifies some representative vulnerabilities and propose assertions for them to show how SoC security assertions can be defined and integrated in an existing SoC validation methodology. Specifically, this chapter makes three major contributions:

1. We perform a comprehensive review of the literature to identify the common SoC security vulnerabilities.
2. We propose security assertions corresponding to each vulnerability.

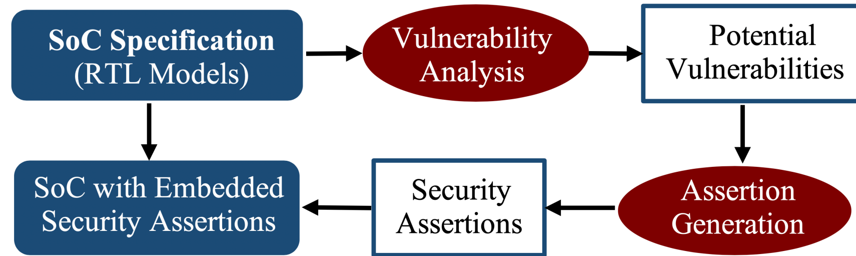


Figure 3-1. The framework for defining and utilizing SoC security assertions.

3. We demonstrate that existing assertion-based validation is not capable of detecting security vulnerabilities.

The remainder of this chapter is organized as follows. Section 3.1 and Section 3.2 provide an overview of assertion-based validation and SoC security vulnerabilities, respectively. Section 3.3 describes the framework for assertion generation for a given set of security vulnerabilities. Section 3.4 presents six case studies. Finally, Section 3.5 concludes the chapter.

3.1 Assertion-based Validation

Assertion-Based Verification (ABV) has been widely used in SoC pre-silicon verification [10]. Synthesized assertions and associated checkers are utilized for post-silicon (as well as runtime) coverage analysis of critical events [82]. Assertions are used to capture the intent of the specification [10]. For example, a functional assertion can check that the output of an adder is equal to the sum of two inputs irrespective of the implementation. In addition to checking the inputs and outputs, assertions can also increase the observability of internal states. Compared to pre-silicon simulation where every signal can be monitored, the observability is very limited during post-silicon validation. As a result, debugging a post-silicon failure can take numerous iterations of trials and errors to localize an error using a small set of observable signals recorded in a trace buffer. Assertions, on the other hand, are able to detect any undesired behavior either from the design or from the environment (e.g., fault injection), and expose the location of errors directly.

3.1.1 Assertion Languages

Temporal logic is powerful in representing assertions, by introducing the notion of timing to propositional logic [83]. There are mainly two types of languages, i.e., Linear Temporal Logic (LTL) [84] and Computational Tree Logic (CTL) [85]. LTL can be used to describe a sequences of transitions between states. The most commonly used operators to describe these transitions are given in Table 3-1 [1], where p and q represent propositions that are either true or false at any given time. CTL allows for path quantifiers, such as E ("there exist a path") or A ("for all paths"). Linear time and path qualifiers can be combined, e.g. EX or AX.

Table 3-1. Commonly used temporal operators in LTL [1]

Operator	Semantics
$X\ p$	("next" state): p is true in the next state of the path.
$G\ p$	("always" or "globally"): p is true at every state on the path.
$F\ p$	("eventually" or "in the future"): p is true at some future state on the path.
$p\ U\ q$	("until"): q is true at some future state, and at every preceding state on path, p is true.

There are mainly two types of approaches for defining hardware assertions: language-based and library-based [86]. Language-based approaches provides syntax for formally defining assertions. Two of the most popular assertion specification languages are Property Specification Language (PSL) [87] and System Verilog Assertions (SVA) [88]. Both of these languages support temporal assertions and formally is an extension of temporal logic [83]. Some other examples are ForSpec [89], SALT [90], a SystemC extension [91], etc. On the other hand, library-based approaches add assertion support to existing languages. One such example is Open Verification Library (OVL) [92]. OVL has support for Verilog, VHDL, PSL and SystemVerilog. Library-based approaches can be used to quickly write common types of assertions. Unfortunately, they are not generic enough to cover all possible scenarios.

3.1.2 Automated Assertion Generation

Assertion generation is mostly manual effort - it is time-consuming to insert enough assertions into an industrial SoC design. Many research efforts have been devoted to automated generation of functional assertions. Rogin et al. [93] proposed to generate

properties of a design by analyzing simulation traces. Hertz et al. [14] improved the analysis process using data mining and developed a tool named Goldmine. The generated rules from simulation traces are passed through a formal verification tool to verify the correctness in the design. As the simulation data is inherently incomplete and nondeterministic, the quality of mined assertions cannot be guaranteed. Moreover, these functional assertions are not suitable for detecting security vulnerabilities.

3.2 SoC Security Vulnerabilities

There is significant prior effort in classifying software-level vulnerabilities [20, 94]. While a lot of work has been done in hardware Trojan detection, it represents only a small fraction of the SoC vulnerability space. Specifically, it belongs to one of the seven classes (the last category) of vulnerabilities outlined in this section. We reviewed a wide variety of security vulnerabilities listed in the National Vulnerability Database [20], and developed the following seven classes of vulnerabilities that are related to SoCs. Note that there is a fundamental difference between exceptions and security vulnerabilities. The exceptions are defined today by SoC designers based on the point of view of functional correctness, whereas the security vulnerabilities outlined in this proposal are solely from security and trust perspectives. For example, a vulnerable design may trigger (e.g., using a Trojan) an exception (e.g., divide by zero) even when the event does not happen. As exceptions are normally handled in a higher privilege level, the user would get access to the registers that should not be granted. The remainder of this section describes our proposed seven vulnerability classes.

3.2.1 Permissions and Privileges

Permissions and privileges are the main components of the access control subsystem. Specifically, different resources are controlled by different permissions and privileges. For example, in ARM7 processor, seven different modes are defined, such as user mode, interrupt mode, and supervisor mode. It is critical to check whether the conditions for triggering privileged modes are satisfied before changing modes.

3.2.2 Resource Management

Certain resources should be protected against any illegal access, including accessing special hardware from non-privileged modes, misuse of design-for-debug infrastructures during normal usage, and so on. For example, JTAG allows engineers to trace secure memory during post-silicon validation and debug of security features. However, JTAG should never be enabled during normal usage.

3.2.3 Illegal States and Transitions

The behavior of an SoC can be modeled as a finite state machine (FSM). The valid states or transitions can be verified during functional validation. Attackers are more interested in the backdoor that allows undefined states/transitions. To verify the existence of illegal states and transitions, we can use both the assertions of the valid states and transitions to show the violation, and enumerate the invalid states and transitions to show the existence of specific vulnerabilities.

3.2.4 Buffer Issues

Modern SoCs consist of advanced features (e.g., out-of-order execution and speculative execution) as well as a large number of heterogeneous buffers. Similar to software buffer errors, these buffers in deeper pipelines require significant validation efforts to detect any remaining flaws. For example, prefetched instructions in buffers should be flushed if the branch prediction is incorrect. Otherwise, these flaws can be exploited to mount an attack.

3.2.5 Information Leakage

Modern SoCs provide the isolation between a secure world and a non-secure world. Information from the secure world should never be leaked to non-secure world directly. ARM uses TrustZone as an approach to provide the secure world [95]. There should be safeguards present to prevent non-secure world from accessing TrustZone directly.

3.2.6 Numeric Exceptions

Numeric exceptions represent the erroneous/illegal behaviors (e.g., divide by zero) during arithmetic computations. Even if the program does not lead to illegal numeric computation, an attacker can make it happen, and utilize it to create a vulnerability.

3.2.7 Malicious Implants

In software community, code injection means allowing attackers to run arbitrary code. Similarly, hardware Trojans, inserted by untrusted third party, allows attackers to execute an arbitrary path after applying specific input patterns. This can lead to information leakage or other unintended consequences. Trojans are usually inserted in hard-to-detect and rare-to-activate areas, making it hard to detect them during validation.

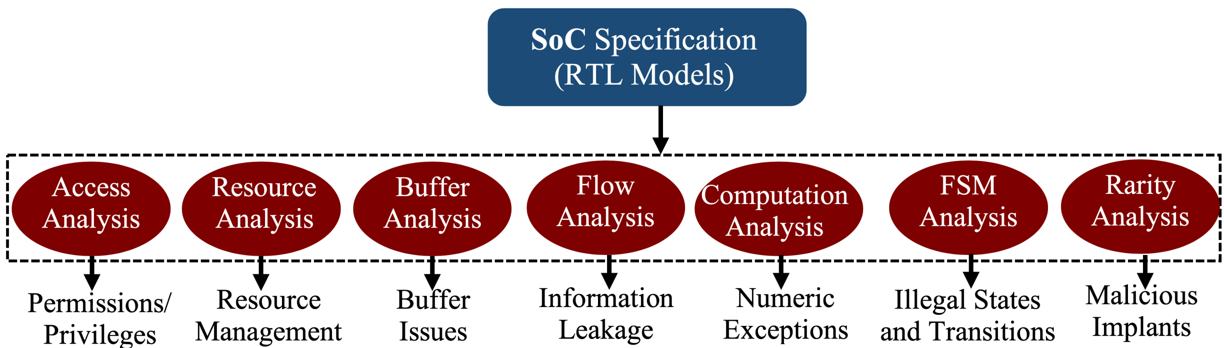


Figure 3-2. Overview of our assertion generation framework for different classes of vulnerabilities.

3.3 SoC Security Assertions

This section mainly addresses two important questions: i) how to generate the security assertions, and ii) how to embed them in the SoC design (RTL models).

3.3.1 Embedding of Security Assertions

There are two orthogonal ways of embedding assertions in the RTL model of the SoC design: immediate and concurrent assertions. Please note that immediate assertions can be converted to concurrent assertions by modifying the antecedent. However, as described below, it would be natural to use a specific one depending on the type of security vulnerability.

Immediate Assertions: Immediate assertions are powerful in detecting vulnerabilities such as numeric errors. Immediate assertions are flexible and can vary based on the potential statements or blocks. For immediate operations, it is important to find out the exact location, the relevant variable (e.g., trigger) α, and the assertion, *assert* $(P(\alpha))$, can be inserted. Immediate assertions are inherent for checking specific operations, such as divide-by-zero checking and out-of-boundary checking.

Concurrent Assertions: Concurrent assertions, on the other hand, are checked each clock cycle, representing expected properties of SoCs. Concurrent assertions are useful to express any FSM related vulnerabilities (e.g., illegal states and transitions). Each concurrent assertion can be defined using *assert property* (P). The property P should be derived from the specification of SoCs and vulnerability classes.

3.3.2 Generation of Security Assertions

We use static (code) analysis to generate security assertions to detect the existence of the vulnerabilities outlined in Section 3.2, as shown in Figure 3-2. In this section, we briefly outline the assertion generation for each vulnerability class.

Permissions and Privileges: By analyzing the specification, we need to figure out the variable that represents the privilege level, e.g., CPSR in ARMv7. For each entry to a privileged operation block, we need to generate an assertion. For the ease of illustration, we use `user` to represent current privilege, and `admin` to represent root privilege. For each possible entry into the privileged operation block, we need to generate the immediate assertion as: $assert($`user == admin`$)$.

Resource Management: Concurrent assertions are powerful in protecting resources from misuse in an unexpected way. For example, to protect JTAG from getting used during normal operation mode, we need to generate the assertion as: *assert property* (`normal` $|->$ `!JTAG_enable`).

Illegal States and Transitions: The behaviors of modules as well as their interactions (protocols) can be expressed in FSMs. Therefore, we can express both valid and invalid (illegal) transitions as security assertions. For example, if there is a valid transition from state A to state B when variable a is true, it can be encoded as an assertion: *assert property (A && a* $|->$ *B)*. Similarly, the assertion, *assert property* $(!(C|->A))$ can be used to ensure that no transitions are allowed from state C to state A.

Buffer Issues: Assertions can be generated for all boundary cases related to buffers. To prevent access of the buffer index beyond its limits, immediate assertions should be added before each buffer access. Before accessing `Buffer[index]`, the variable `index` needs to satisfy *assert* `(index >= 0 && index <= limit)`. In many scenarios, we may require concurrent assertions to ensure global interactions. For example, to ensure the flush of instruction buffer (IB) after branch prediction failure, we can use the assertion *assert property* `(Pre_fail` $|->$ `IB_Empty)`.

Information Leakage: To protect secret information from directly leaking to non-secure world, tagging is one potential solution. It assumes that the results of secure world can only be passed to non-secure world through special interface (privileged instructions). For each normal operation consisting of both secure and non-secure variables, we need to check if the result is assigned to a non-secure variable as expected. If s is a secure variable, the assignment of s to a variable v needs to check the tag of v as *assert(v_t* $==$ *secure_tag)*.

Numeric Exceptions: Numeric exceptions are more relevant to the implementation of SoC designs. We need to generate one assertion for each possible numeric exception during arithmetic computation. For example, in case of a divide-by-zero exception, we can generate an immediate assertion as: *assert*`(divisor != 0)`.

Malicious Implants: Malicious modifications (e.g., hardware Trojans) can be inserted during pre-silicon or post-silicon stage. In the pre-silicon stage, hardware Trojans are usually hidden in rare-to-activate branches or rare execution of concurrent statements. For example, we can

generate assertions for each rare branch by adding an assertion for each rare trigger condition as: *assert(rare_trigger)*.

The security assertions are generated based on the vulnerabilities outlined in Section 3.2. More assertions can be added based on designer inputs or application-specific considerations.

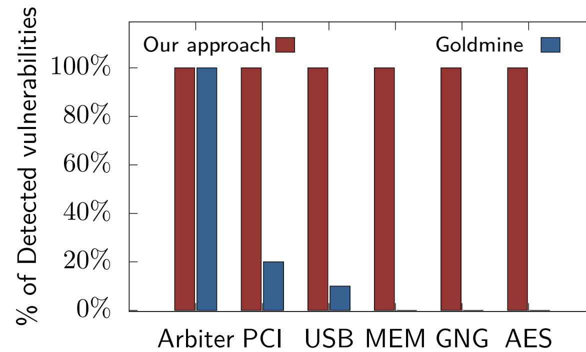


Figure 3-3. Comparison of detected vulnerabilities by our assertions and Goldmine [14].

3.4 Case Studies

To demonstrate the necessity of security assertions, we analyzed six benchmarks and inserted security assertions introduced in Section 3.3. Then, Goldmine [14] was applied on all the benchmarks to generate as many assertions as possible. To evaluate the effectiveness of our security assertions, we randomly inserted 10 vulnerabilities into the design to form 10 vulnerable instances and applied directed test to activate these vulnerabilities. If any assertion generated by the two methods (ours versus Goldmine) got activated during simulation, we claim the corresponding method detects the vulnerability. The types of potential vulnerabilities of each benchmark and the detected instances are shown in Table 3-2 and Figure 3-3, respectively. Note that the number of instances are more than the number of vulnerability types, as each type may contain multiple instances. In the remainder of this section, we describe each type of vulnerability and inserted instances in detail. Overall, our approach is able to detect all of the vulnerabilities while the assertions generated by Goldmine fail to detect most of them.

Table 3-2. Types of vulnerabilities explored in the six benchmarks.

Vulnerability	Arbiter	PCI	USB	MEM	GNG	AES
Permissions and Privileges				✓		
Resource Management			✓			✓
Illegal States & Transitions	✓	✓	✓			
Buffer Issues				✓		
Information Leakage				✓		
Numeric Exceptions					✓	
Malicious Implants						✓

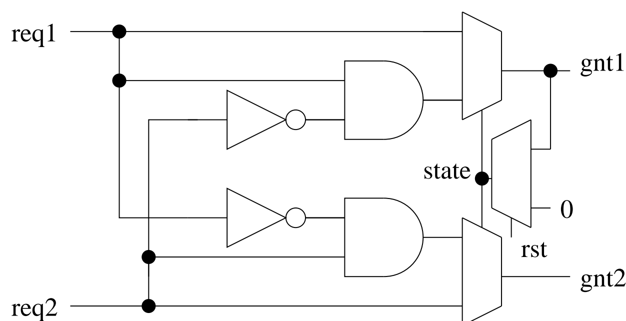


Figure 3-4. A simple arbiter with four inputs (clk not shown) and two outputs.

3.4.1 Arbiter

We first analyzed a simple design, Arbiter, as shown in Figure 3-4. For this simple design, we inserted the vulnerability of **invalid states and transitions**. Even for this small design, careless design, fault injections, or transient errors can make it behave differently. For example, if the security of whole design relies on the *gnt1* and *gnt2* not asserted together, *assert*(!(gnt1 & gnt2)) should be added to the design. However, the number of invalid states and transitions would be exponential when time is involved. For example, when we consider two consecutive cycles, *assert*(!(gnt1 | => (gnt2 != req2))) should hold. We limit the number of assertions to 10 in this example.

The assertions generated by Goldmine are shown in Listing 3.1. As Goldmine analyzes the golden design by mining the traces of random simulation, it is able to capture reachable states and transitions by the specific simulation.

Listing 3.1. Assertions of Arbiter by Goldmine

```
(state == 1 & req2 == 1) |-> (gnt1 == 0)
(req1 == 1 & state == 0) |-> (gnt1 == 1)
(req1 == 0) |-> (gnt1 == 0)
(req1 == 1 & req2 == 0) |-> (gnt1 == 1)
(req1 == 1 & state == 0) |-> (gnt2 == 0)
(req2 == 1 & state == 1) |-> (gnt2 == 1)
(req2 == 0) |-> (gnt2 == 0)
(req2 == 1 & req1 == 0) |-> (gnt2 == 1)
```

The 10 vulnerability instances are generated by mimicking the behavior of fault injection, i.e., randomly inverting one signal in Figure 3-4. From Figure 3-4, we can see that Goldmine is good at detecting vulnerabilities involving finite state machines in this small design.

3.4.2 PCI

Top module pci_master32_sm from Opencores [96] contains eight modules such as pci_frame_crit and pci_irdy_out_crit. To mimic SoC design, which contains different parts from untrusted third party, we inserted **invalid states and transitions** to the subordinate modules (8 out of 10) as well as the top modules (2 out of 10). Goldmine generated 19 assertions. To generate vulnerable instances, we randomly changed operators in all the modules. As shown in Figure 3-3, Goldmine was able to capture the two vulnerabilities, but failed to detect the remaining eight vulnerabilities.

3.4.3 USB Protocol

USB protocol defines the packet fields and its corresponding operations. We analyzed the USB protocol module usbf_pa.v along with CRC module from Opencores [96] and identified two types of vulnerabilities. The USB protocol depends on the packet ID (PID) to identify the types of packets including token, data, handshake, and special. For each type of packets, PIDs will not overlap. In the module, the output of PID is stored in tx_data whose value depends on the input selectors. The two types of vulnerabilities are shown below:

1. **Resource management:** As the PIDs define the resources of USB, it is critical to make sure that each type of packet gets expected PID. For example, a token packet should not get any of the PIDs belonging to data packets, such as DATA0, DATA1. Similarly, a handshake packet should stick to its own type, e.g., ACK, NAK. We inserted 8 assertions of this type.

2. **Invalid states and transitions:** We inserted one vulnerability in CRC module and one vulnerability in the top module. As this is a common vulnerability in almost every design, we will skip inserting this type of vulnerability in the remaining benchmarks.

The vulnerable instances were generated accordingly. As shown in Figure 3-3, while Goldmine can only detect one vulnerability from state transition in top module, our assertions can detect all of them.

3.4.4 A Simplified Memory Design

This design is created to mimic the behavior of a simplified Trusted Hardware (TH) implementation of memory, as shown in Listing 3.2. Trusted Hardware, e.g., Intel's Software Guard Extensions (SGX) [97], allows remote clients to upload private computation and data to a secure container of a server with a TH. One key implementation of SGX is the introduction of Process Reserved Memory and Enclave Page Cache (EPC), inhibiting invalid accesses even from the kernel. The simplified design is shown in Listing 3.2 which contains input signal *sc* to denote whether it is a secure access or not. The memory space is denoted by an array named *mem* with size of 1MB.

1. **Permissions and privileges**. Assume the lower 1kB of memory is allocated to EPC. Since EPC should be accessed through secure container/process, each access to the lower 1kB should be checked. Although one conditional checking is already in place, assertions may also help when implementation error, fault injection or Hardware Trojans exist, e.g., *assert*(address $<=$ 1024 $|$ $=>$ sc) can be inserted before any access to memory.

2. **Information leakage**. In this simplified memory implementation, it does not explicitly describe the state of *out* signal when we want to write. For a buggy CPU design which connects to this memory, a process may be able to read the previous access of another process from the out port (including secure processes) with interleaved memory access. We may add a concurrent assertion with (*assert property* (wr $|$ $=>$ out $==$ 0)).

3. **Buffer errors**. Memory as one type of buffer should be checked for buffer errors. Each access to memory should be checked with assertions to test if address is in the range of

memory size. In this example, the memory size is 1MB (2^{20}) and the length of address is 20 bits. We need *assert*(address $< 2 * *20$) to avoid errors such as careless typo.

Listing 3.2. Simplified Memory of Trusted Hardware

```
module mem(clk, rst, wr, sc, address,
           in, out);
input clk, rst, wr, sc;
input [19:0] address;
input [7:0] in;
output reg [7:0] out;
reg [7:0] mem[2**20-1:0];
always @ (posedge clk)
  if (address >= 1024 || sc) begin
    if (wr) mem[address] <= in;
    else out <= mem[address];
  end
endmodule
```

For the permissions and privileges, we inserted a vulnerability by removing *sc* checking in the first *if* statement. For the buffer errors, we assume a typo in the length of address definition. The remaining vulnerabilities are about information leakage. We assume that attackers are able to connect one specific location to *out* when it is a write operation. Experimental results show that Goldmine failed to detect any of these vulnerabilities while our approach can detect all of them.

Listing 3.3. GNG_interp

```
module gng_interp (
  input clk, rstn, valid_in,
  input [63:0] data_in,
  output reg valid_out,
  output reg [15:0] data_out
);
wire [33:0] mul1;
wire signed [13:0] mul1_new;
reg [17:0] c0_r5;
reg signed [18:0] sum2;
reg [14:0] sum2_rnd;
assign mul1_new = mul1[32:19];
always @ (posedge clk)
  sum2 <= $signed({1'b0, c0_r5}) + mul1_new;
always @ (posedge clk)
  sum2_rnd <= sum2[17:3] + sum2[2];
...
endmodule
```

3.4.5 Gaussian Noise Generator (GNG)

We next inspected a computation-intensive design called Gaussian Noise Generator (GNG). The design is downloaded from Opencores [96]. One possible **numeric error** in this design is the assignments between signed and unsigned values as shown in the snippet of code in Listing 3.3. The first assignment assigns an unsigned variable to a signed variable. Next assignment is the computation between signed values. The final assignment assigns a signed value to an unsigned value. We are concerned with the automatic transformation between signed and unsigned values. For example, when the 32th bit of mul1 is 1, mul1_new

is interpreted as a negative value using a two's complement representation. Then mul_new is added to a positive number and converted to an unsigned number again. The behavior may or may not be the original intention of this code. We want to generate assertions, e.g., *assert*(mul1[32] != 1), and guide the test plan of debug. The developer should decide if it is a numeric error or the expected behavior. Since we view this design as "buggy" by itself, we did not generate vulnerable instances for it. Rather, we use 10 direct tests to force mul1[32] to become 1 and check if the assertions by Goldmine can catch the potential vulnerability. As shown in Figure 3-3, Goldmine failed to detect any of them since it only analyzes the traces of random simulation but never inspects the specification/implementation.

3.4.6 AES

Advanced Encryption Standard (AES) is a very commonly used crypto core, consisting of ten rounds of block ciphers (substitution permutation networks). The substitution is shown in Listing 3.4. We also inserted JTAG to dump internal variables during debug. The identified vulnerabilities are:

1. **Resource management:** As JTAG is for debug purpose only, it should be disabled during normal usage. As a result, the dump signal should contains nothing related to any internal signals. We inserted an concurrent assertion *assert property* `(!JTAG |-> (JTAG_out != in))` to prevent attackers from bypassing the JTAG checking and dump internal signals to infer the plaintext.

2. **Malicious implants:** As module S contains a lot of rare branches, attackers are able to construct rare trigger conditions for hardware Trojans. For example, the probability of (out == 32'h7c7c7c7c) is 2^{-32} when (in == 8'h01) is true for all S_0, S_1, S_2 and S_3 together. Assertions like *assert*(out != 32'h7c7c7c7c) in module S4 can guide the designer of a test plan to cover this specific potential trigger condition. As the combinations of rare branches are potentially infinite, we restricted the number of assertions to be 10.

One of our vulnerable instances is a design bypassing JTAG checking directly. For the other 9 instances, we construct random hardware Trojans from the rare branches. As shown in Figure 3-3, our approach is able to detect the vulnerabilities. Goldmine cannot detect any of them.

Listing 3.4. AES table

```
module S4 (clk, JTAG, in, out, JTAG_out);
  input clk, JTAG;
  input [31:0] in;
  output [31:0] out;
  output reg [31:0] JTAG_out;
  S
    S_0 (clk, in[31:24], out[31:24]),
    S_1 (clk, in[23:16], out[23:16]),
    S_2 (clk, in[15:8],  out[15:8] ),
    S_3 (clk, in[7:0],   out[7:0]  );
  always @ (posedge clk)
    if (JTAG) JTAG_out <= in;
endmodule
module S (clk, in, out);
  always @ (posedge clk)
  case (in)
    8'h00: out <= 8'h63;
    8'h01: out <= 8'h7c;
    ...
  endcase
endmodule
```

Overall, the security assertions generated by our approach are able to detect all the security vulnerabilities whereas the assertions generated by state-of-the-art technique (Goldmine) fail to detect most of them.

3.5 Summary

SoCs are widely used today in both embedded systems and IoT devices. While SoC security is paramount, there are no prior efforts in defining and detecting a wide variety

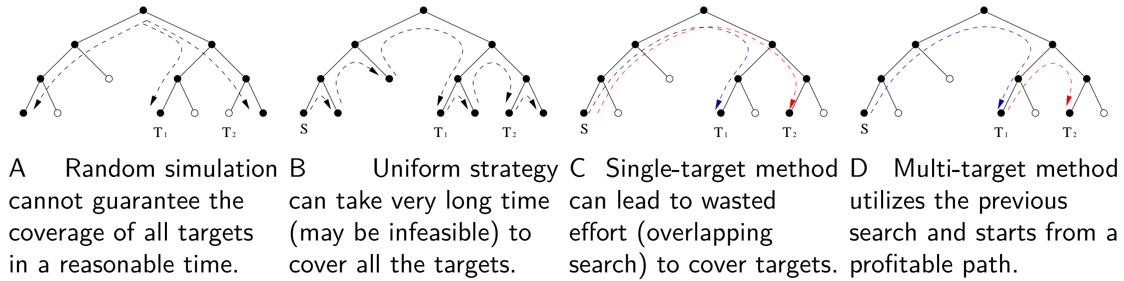
of SoC security vulnerabilities. In this chapter, we developed seven classes of SoC security vulnerabilities. Based on these vulnerabilities, we proposed a framework for generating security assertions. Using a diverse set of benchmarks, we demonstrated that the functional assertions generated by state-of-the-art assertion generation technique cannot eliminate the need for our dedicated security assertions. Specifically, our security assertions are able to detect all the implanted security vulnerabilities while the state-of-the-art method failed to detect most of them. We envision that the SoC designers will embed security assertions in their designs in the near future as part of their assertion-based security validation methodology.

CHAPTER 4
SCALABLE CONCOLIC TESTING OF RTL MODELS

Simulation is widely used for validation of Register-Transfer-Level (RTL) models. While simulating with millions of random or constrained-random tests can cover majority of the targets (functional scenarios), the number of remaining targets can still be huge (hundreds or thousands) in case of today's industrial designs. While directed test generation techniques using formal methods are promising in such cases, it is infeasible to apply them on large designs due to state space explosion. The application of concolic testing on hardware designs has shown some promising results in improving the overall coverage. However, existing concolic testing approaches are not designed to activate specific targets such as uncovered corner cases and rare functional scenarios. In other words, these approaches address state space explosion problem but lead to path explosion problem while searching for the uncovered targets. Uniform test generation [2] tries to maximize the overall branch/statement coverage but ignores the priority of activating specific targets, leading to longer test generation time, as shown in Figure 4-1B. As the number of paths grow with unrolled cycles, uniform test generation will suffer from path explosion problem and will not be able to finish within the time limit. Ahmed *et al*. [98] proposed a promising approach to guide path exploration to reach a specific target. As demonstrated in Section 4.2.2, their approach leads to exploration of undesired paths in many scenarios.

In this chapter, we propose a fully automated and scalable approach for generating directed tests using concolic testing of RTL models. Our proposed approach maps directed test generation problem to target search problem while avoiding overlapping searches involving multiple targets. We first propose an efficient path exploration scheme to improve the quality of explored paths and coverage of a specific target. We call it *single-target method*. However, single-target method is not suitable for scenarios when thousands of targets (not covered by millions of random simulation) need to be activated since a lot of effort is wasted in overlapping searches, as shown in Figure 4-1C. To reduce the number of overlapping searches,

we propose efficient learning and clustering techniques activate multiple targets. Our approach utilizes information from the previous searches, as shown in Figure 4-1D.



A Random simulation cannot guarantee the coverage of all targets in a reasonable time.
B Uniform strategy can take very long time (may be infeasible) to cover all the targets.
C Single-target method can lead to wasted effort (overlapping search) to cover targets.
D Multi-target method utilizes the previous search and starts from a profitable path.

Figure 4-1. Comparison of four approaches in covering two targets T_1 and T_2.

In this chapter, we make two major contributions:

1. We propose a scalable test generation technique using concolic testing of RTL models to activate a specific target. We develop a novel contribution-aware edge realignment technique to effectively evaluate the distance between a simulated path and a specific target. The realigned edges are used to guide alternative branch selection to improve both functional coverage and test generation efficiency. Based on the realigned edges, we propose a path exploration algorithm to quickly activate the targets by searching "close" alternative branches.
2. In order to exploit learning across test generation instances involving multiple targets, we explore two optimization techniques to effectively utilize previous search results. We utilize target pruning to eliminate targets that are covered by the tests generated for activating other targets. We also minimize the overlapping search efforts by employing clustering of related targets to drastically reduce the overall test generation time.

The remainder of the chapter is organized as follows. The overview of our framework is outlined in Section 4.1. Section 4.2 presents our proposed test generation approach for activating a specific target. Section 4.3 describes various optimizations for activating multiple targets. Section 4.4 presents experimental results. Finally, Section 4.5 concludes the chapter.

4.1 Overview and Problem Formulation

Given an RTL description of a hardware design, our proposed approach will generate a set of compact tests to cover all the hard-to-activate branch targets. This section is organized as follows. We first describe the modeling of targets. Next, we provide an overview of our proposed approach and outline the organization of the remainder of this chapter.

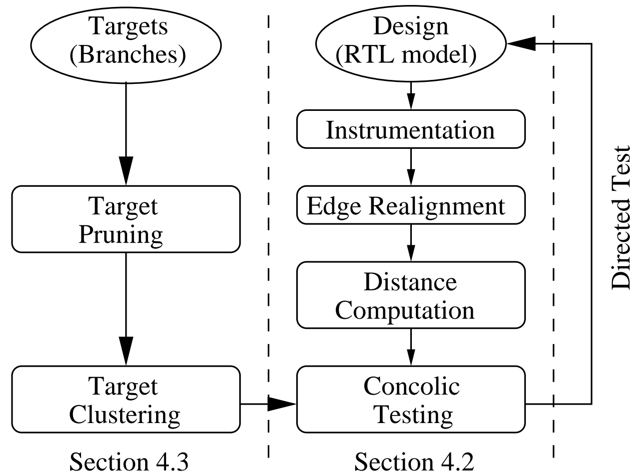


Figure 4-2. The overview of our test generation framework using concolic testing.

4.1.1 Modeling of Targets

In this chapter, a validation target (**target**, in short) in RTL model represents a branch condition that the verification engineer would like to cover during validation of RTL models. Specifically, we are interested in the hard-to-activate branches in a traditional simulation-based validation methodology. In addition to branch target in the design, our approach can also be used to validate other scenarios that can be converted to equivalent branch statements, such as assertions [99].

4.1.2 Overview

Figure 4-2 shows the overview of our framework. We first simulate the design with random test vectors to find out the hard-to-activate branch targets. To activate a specific target, we apply three steps to preprocess the RTL code, i.e., instrumentation, edge realignment and distance computation as shown in Figure 4-2. The instrumentation is to provide simulation information to symbolic solver by adding print statements to the design. The goal of edge realignment is to directly connect each basic block to the assignments that control the entry condition to this block. After edge realignment, distance computation is able to determine the most profitable alternative branches to activate a target. In summary, the main goal of preprocessing is to assist alternative branch selection in concolic testing (see Figure 2-2). The details are discussed in Section 4.2. For multiple targets, we apply target

pruning and target clustering to reduce overlapping searches, by pruning targets that are covered during searching paths for other targets, and finding the most profitable initial path for the remaining targets. These optimizations are discussed in Section 4.3. The generated tests are validated in the original design (without instrumentation and edge realignment) to check if the desired targets are covered.

4.2 Test Generation using Concolic Testing

A major challenge in concolic testing is how to efficiently explore profitable paths to activate a specific target. As shown in Figure 4-2, our test generation framework to activate a single target consists of four major steps: instrument the code to add print statements, realign edges to reveal contributions of each assignment, compute distance and explore different paths to cover the target. With the help of edge realignment, we are able to heuristically evaluate the distance between a path and the target. As shown in Figure 4-1, both random and uniform test generation do not consider the quality of a selected path. In contrast, our directed path selection tries to explore paths that are "closer" to a specific target. This section describes these steps in detail.

4.2.1 RTL Code Instrumentation

The first step is to instrument the original design. The goal of instrumentation is to provide information of a concrete path for symbolic execution. There are two possible ways to do symbolic execution. One option is to modify the RTL simulator directly such that symbolic execution is performed along with concrete simulation. The other one is to instrument the original design such that path execution information can be dumped. After simulation is done, the symbolic execution engine will parse and analyze the dumped traces. Our framework utilizes the latter method, since it is more adaptive to different simulators and languages. The simulation traces would be the same irrespective of the simulator. Note that the instrumented RTL code is just for test generation. In our experimental evaluation, the original design is used to verify that the generated tests activate the expected branches.

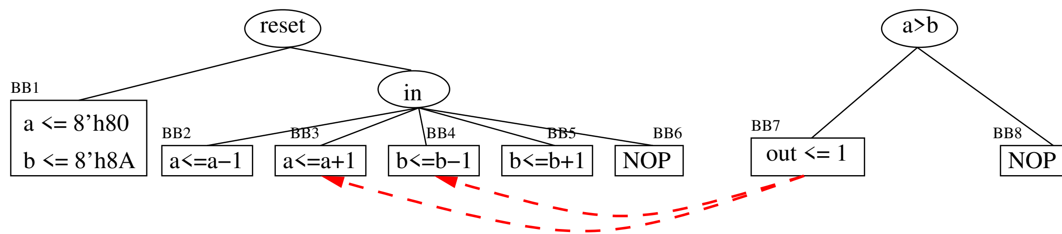
Listing 4.1. Example 1

```
module top(clock, reset, in, out);
reg [7:0] a, b;
reg out = 1'b0;
always @(posedge clock) begin
    if (reset == 1'b1) begin
        a <= 8'h80; b <= 8'h8A;
        $display("BB1");
    end
    else case (in)
        8'h01: a <= a - 1; $display("BB2");
        8'h23: a <= a + 1; $display("BB3");
        8'h45: b <= b - 1; $display("BB4");
        8'h67: b <= b + 1; $display("BB5");
        default:           $display("BB6");
    endcase
end
always @(posedge clock) begin
    if (a > b) begin
        out <= 1'b1; $display("BB7");
    end
    else begin
        $display("BB8");
    end
end
endmodule
```

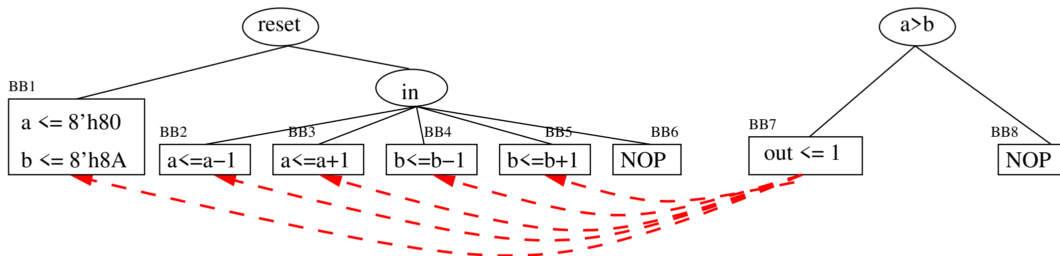
Our framework first parses the design and constructs its control flow graph. Then, it marks each basic block with a unique identifier (BBi for the ith basic block). The unique identifiers of basic blocks help symbolic engine build the constraints which contain all the assignments inside each basic block. Then, we instrument the design by adding a print statement at the end of each basic block. The example of an instrumented design is shown

in Listing 4.1 with the added print statements shown in gray. Its corresponding CFG is shown in Figure 4-3 (without dashed lines). At the same time, our framework generates a testbench module that is able to read the tests generated by our approach and provide the stimuli to the instrumented design. After simulating one input vector, a trace showing the executed basic blocks is dumped and analyzed. For example, with a random input test, it is highly likely to get a trace [BB1, BB8, BB6, BB8, BB6, BB8, ...]. After reconstructing the simulation path from the trace, next step of concolic testing is to choose a new path to explore and generate a corresponding test to exercise the path.

4.2.2 Contribution-aware Edge Realignment



A Our contribution-aware edge realignment. Realign each block to the assignments that contribute to the activation of that block.



B [3] realigns each block to the assignments that are satisfiable.

Figure 4-3. Comparison of edge realignment by our approach and [3].

Our framework tries to explore paths that can take it "closer" to a specific target. Alternative branch selection in concolic testing is essentially "forcing" the next execution path to pass through a specific block. When a "good" alternative branch (block) is selected, the simulation path will get closer to the target. On the other hand, a "bad" alternative branch (block) will lead the simulation trace randomly or far away from the target. The goal of our edge realignment is to figure out the contribution of each block in activating a specific target.

In this section, we propose a contribution-aware edge realignment scheme to enable smart selection of profitable alternative branches while exploring new paths.

efore we describe the details of the process, let us use an example to show the results of edge realignment. Let us consider the example in Listing 4.1 to activate the branch in BB7 with the branch condition of *(a>b)*. We note that the original CFG in Figure 4-3 (without dashed lines) provides no information about which block is good or bad. To manually write a test to activate BB7, a test writer tries all blocks containing the assignments for signals a and b. While this manually backward tracking is viable for small designs, it is not feasible for large designs due to scalability issues. Instead of manually figuring out which basic blocks are relevant in activating BB7, we create reference edges to directly connect the blocks with profitable assignments to our target block. For example, the dashed lines in Figure 4-3(a) connect BB3 and BB4 to our target block BB7. These realigned edges instruct our concolic testing framework to prefer BB3 and BB4 to other blocks in selecting alternative branches. Intuitively, the paths across these two blocks are more likely to activate BB7 than the other blocks.

For the ease of illustration, we use the following notations. We use $\boldsymbol{s}$ to represent a global state, containing a snapshot of the values of all registers and wires in a specific time. Let $g(\cdot)$ be the guard condition of a basic block. For example, the global state $\boldsymbol{s_0} = \{a_0 = 8'h80, b_0 = 8'h8A\}$[1] after executing BB1. The subscript of a variable is used to keep track of the different values during the whole simulation. In other words, the subscript of a variable increments by 1 whenever an assignment to the variable is executed. The guard condition for BB7 is $g(\cdot) = a > b$. If we evaluate the guard condition directly on $\boldsymbol{s_0}$, i.e., $g(\boldsymbol{s_0}) = (a_0 > b_0) = False$, it represents that BB7 cannot be visited right after

[1] For the simplicity of explanation, we only show the relevant variables to our condition g in $\boldsymbol{s}$. For example, when g is $a > b$, we only show the variables a and b. In our framework, all variables are kept in the global state.

executing BB1. We use $f(\cdot) : \boldsymbol{s} \rightarrow \boldsymbol{s'}$ to represent any assignment that may change the global state. For the assignment $f = (a <= a + 1)$, it changes the global state $\boldsymbol{s}_0$ to $\boldsymbol{s'} = f(\boldsymbol{s_0}) = \{a_1 = 8'h81, b_0 = 8'h8A\}$, where only the value of a is changed. For the ease of representation, a sequence of assignments is represented using composition, i.e., $\boldsymbol{s'} = \boldsymbol{s} \circ f_1 \circ f_2 ... \circ f_n$, where the assignments $f_1, ...f_n$ are executed in order. The goal of concolic testing is to find a viable sequence of assignments $f_1, ..., f_n$ that will hit a specific target, i.e., $g(\boldsymbol{s'}) = True$.

A naive way to realign edges was proposed in [3] by simply checking the satisfiability of one assignment and the guard condition of a block. For example, assuming the guard condition of a target is $(v \,\&\, 0 == 0)$, which requires v to be an even number, the naive edge realignment will realign the target to all blocks where v is possibly assigned an even number. While this approach is promising in connecting simple conditions to a single assignment, the naive checking introduces a large number of redundant edges which can mislead path selection. The result of applying the naive edge realignment to Listing 4.1 is shown in Figure 4-3(b). Compared to our expected results in Figure 4-3(a), the naive approach has two major problems. The first one is that the naive approach lacks the checking of contribution. The naive edge realignment scheme in [3] connects the target to all satisfiable assignments of its variables, e.g., BB7 is connected to BB2 in Figure 4-3(b). It is due to the satisfiability of the guard condition $a > b$ and the assignment $a <= a - 1$, i.e., $((a_1 == a_0 - 1)$ *and* $(a_1 > b_0))$ is satisfiable. However, it is easy to see that selecting this assignment is not profitable in achieving the target BB7. When $a_0 > b_0 + 1$ before the assignment is executed, the target BB7 is already activated. Otherwise, the assignment will lead the search path to be far away from BB7. The second problem is the level of satisfiability checking. The naive edge realignment checks the contribution of each individual assignment, rather than all assignments inside a block. For example, assignment level satisfiability checking will connect BB7 to the block containing the assignment $a <= 8'h80$ (BB1). However, when we consider all the assignments inside BB1 together, it is clear to see that executing BB1 will never help to activate the target

BB7. While it is possible to manually check the contribution for small designs, it is infeasible when the design is large and the condition is complex. We propose a contribution-aware block-level edge realignment in Algorithm 1.

Algorithm 1 Edge Realignment

```
 1: procedure REALIGN(CFG, Target Queue (TQ))
 2:     Push all targets to block queue BQ
 3:     while BQ is not empty do
 4:         Current block, bb ← BQ.pop()
                                                ▷ Update edge for block bb
 5:         g ← expanded guard condition of bb
 6:         for variables v ∈ g do
 7:             for assignments f to v do
 8:                 bb′ ← the block of f
 9:                 f_1, f_2, ..., f_n ← all the assignments of bb′
10:                 if g(s) = False and g(s ∘ f_1 ∘ f_2 ∘ ... ∘ f_n) = True for any s then
11:                     Add bb′ to bb.predecessors
12:                     BQ.push(bb′) if bb′ is not visited
13:                 end if
14:             end for
15:         end for
16:     end while
17:     return Realigned CFG
18: end procedure
```

In this algorithm, the block queue BQ maintains all the basic blocks that need to be aligned. Initially, BQ contains all the targets. We first expand the guard condition g for the current block bb to get all the related variables. Then we check all the assignments that are related to any of these variables. For each of these assignments f, we first find out the basic block bb' that contains f. Then, we evaluate its contribution to the guard condition g based on Definition 4.1.

Definition 4.1. *A basic block B has a* contribution *to a guard condition g, if there exists some initial global state $\boldsymbol{s}$, such that $\boldsymbol{s}$ does not satisfy the guard condition g, but the state after executing all assignments inside B satisfies the guard condition. Assume that $f_1, f_2, ..., f_n$ are the assignments inside B. The contribution of B to the guard condition g can be checked by the satisfiability, $g(\boldsymbol{s}) = False$ and $g(\boldsymbol{s} \circ f_1 \circ f_2 \circ ... \circ f_n) = True$.*

Contribution checking forces guard condition g to be false in the beginning, followed by executing all the assignments inside a basic block B, and then checks if g will be satisfied. If it is satisfiable, the block B has a contribution to g. For example, the block BB3 contains only one assignment $f = (a <= a + 1)$. It has a contribution to the guard condition $g = (a > b)$, because $((a_0 \leq b_0)$ *and* $(a_1 = a_0 + 1)$ *and* $(a_1 > b_0))$ has at least one solution. On the contrary, the block BB2 with the assignment $f = (a <= a - 1)$ has no contribution to the guard condition, as the satisfiability equation $((a_0 \leq b_0)$ *and* $(a_1 = a_0 - 1)$ *and* $(a_1 > b_0))$ has no solution. Similarly, BB1 has no contribution since $((a_0 \leq b_0)$ *and* $(a_1 = 8'h80)$ *and* $(b_1 = 8'h8A)$ *and* $(a_1 > b_1))$ has no solution. The results of satisfiability checking for the block BB7 are shown in Table 4-1.

Table 4-1. The results of satisfiability checking in line 10 of Algorithm 1 for the target BB7.

Block	Equation	SAT
BB1	$(a_0 \leq b_0) \wedge (a_1 = 8'h80) \wedge (b_1 = 8'h8A) \wedge (a_1 > b_1)$	UNSAT
BB2	$(a_0 \leq b_0) \wedge (a_1 = a_0 - 1) \wedge (a_1 > b_0)$	UNSAT
BB3	$(a_0 \leq b_0) \wedge (a_1 = a_0 + 1) \wedge (a_1 > b_0)$	SAT
BB4	$(a_0 \leq b_0) \wedge (b_1 = b_0 - 1) \wedge (a_0 > b_1)$	SAT
BB5	$(a_0 \leq b_0) \wedge (b_1 = b_0 + 1) \wedge (a_0 > b_1)$	UNSAT

After finding a good block bb', we add it to the predecessors of bb, i.e., creating an edge to connect bb' and bb. For example, as the block BB3 has a contribution to $a > b$, it is added to the predecessors of BB7. Since BB3 is not visited before, it will be added to the end of BQ. In some future iteration, BB3 will be selected as the current block, and our algorithm will realign good assignments for its guard condition. It is easy to see that Algorithm 1 is fast since no basic block needs to be visited more than once. A good edge realignment scheme is important because even a single bad realigned edge will waste a lot of searching time in finding good alternative paths during path exploration, and it can lead to a wrong direction, which will be demonstrated in Section 4.4.5.

4.2.3 Distance Computation

Edge realignment connects a target to the blocks that have direct contributions. To quantify the contribution of all blocks, we use a distance measurement based on the realigned

control flow graph. A block with lower distance means it is closer to the target, i.e., more likely to contribute to the activation of the target.

First, we define the distance between a basic block and the target. With the realigned CFG in Figure 4-3(a), we start from our target BB7 and perform breadth-first traversal in the direction along the predecessors. For BB7, we initialize the distance as 0, and increment the distance by 1 when we traverse an edge. The distances of the basic blocks in the first *always* block of Listing 4.1 are shown in Figure 4-4, which is unrolled for three cycles. Note that the distances of basic blocks that are never visited are not shown, which will be initialize to ∞ in our framework. For each basic block bb, the distance of bb is denoted as $bb.distance$. Next, we define the distance between a path and the target in Definition 4.2.

Definition 4.2. *Assume a simulation path is constructed by a trace* $\{\{bb_1^1, bb_2^1, \ldots, bb_{i_1}^1\}$, $\{bb_1^2, \ldots, bb_{i_2}^2\}$, ..., $\{bb_1^k, \ldots, bb_{i_k}^k\}\}$, *where* bb_i^j *represents the* i^{th} *basic block in* j^{th} *clock cycle. The distance between the path and a target is the minimum distance among all blocks, i.e.,* $\min_{i,j} bb_i^j.distance$.

Figure 4-4 shows the example of three paths. As all the blocks along P1 and P3 have the distance ∞, their distances to the target are ∞. On the other hand, as P2 passes through BB3 in the second clock cycle, the distance of P2 is 1. When we inspect the final state after executing these three paths, P2 is "closer" to the target, since P2 executed one more $a <= a + 1$. In other words, the distance is a good quality indicator of a path. This distance definition also emphasizes the importance of good edge realignment schemes. If we realign the CFG using the naive approach (as shown in Figure 4-3(b)), the path P3 will have distance 1, same as P2. However, since P3 executed $a <= a - 1$, the final state is actually further away from our target. These realigned edges will mislead directed path exploration as described in the next section.

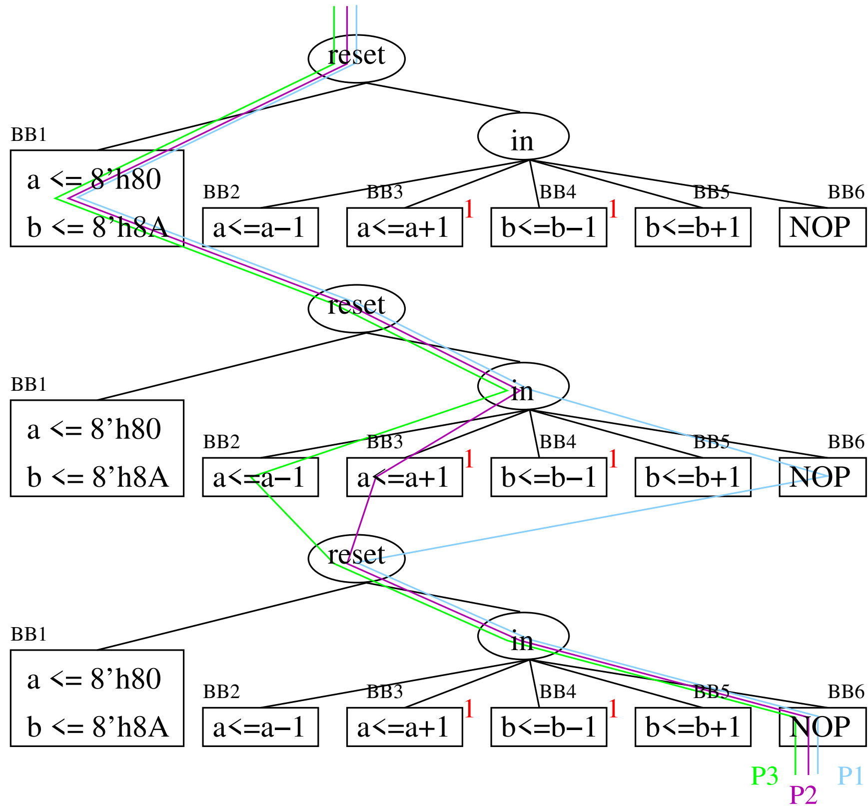


Figure 4-4. The distance between a basic block and the target in realigned CFGs.

4.2.4 Path Exploration

In this section, we present a greedy path exploration scheme in activating a specific target based on our distance heuristic. We illustrate the usefulness of distance information in our automated path exploration scheme.

Algorithm 2 describes our greedy path exploration to quickly reach a specific target by selecting the most profitable alternative branch. To better illustrate how we select alternative branches and explore new paths, we use the example in Figure 4-4. It first computes the distance for all blocks as introduced in Section 4.2.3. Assume that the initial path p is P1. Next, it builds constraints vector from the simulation trace of p, $S = \{\{a_1 = 8'h80, b_1 = 8'h8A\}, \{\}, \{\}\}$, where each inner vector represents all executed statements one clock cycle.

Algorithm 2 Path Exploration

```
1: procedure PATHEXPLORE(realigned CFG, Target Queue TQ, search limit limit, unrolled
   cycles unroll)
2:     for target ∈ TQ do
3:         Compute the distance from target for all blocks
4:         Random simulation and get the path p
5:         iteration = 0
6:         clock = 0
7:         while iteration < limit do
8:             Build constraints vector S from the trace of p
9:             AB ← all valid alternative branches (blocks) after cycle clock
10:            Sort AB by distance and clock
11:            Randomly choose one of the best alternative branches (blocks) to flip
12:            clock ← the clock of the chosen branch
13:            Build the new constraints vector
14:            Use a constraint solver to solve the constraints
15:            Simulate the design with returned test and get a new path p
16:            iteration = iteration + 1
17:            if p activates the target then
18:                Add the test to T
19:                Break
20:            end if
21:            if clock == unrolled then
22:                Increment the distance of all blocks in p
23:                clock = 0
24:            end if
25:        end while
26:    end for
27:    Return A test set T = {t_1, t_2, ..., t_n}
28: end procedure
```

Then, we try all alternative branches[2] from current path p and check if they are viable. For example, we want to check if BB7 is a valid block in clock 2. A new constraints vector is built by combining all constraints before BB8 in clock 2 and the new chosen branch, i.e., $\{\{a_1 = 8'h80, b_1 = 8'h8A\}, \{a_1 > b_1\}\}$. As it is not satisfiable, BB7 is not a valid block in clock 2. On the other hand, BB2 in clock 2 is a valid block, as $\{\{a_1 = 8'h80, b_1 =$

[2] We assume the first clock is only used to reset all signals. Therefore, we will skip the first clock cycle in finding alternative branches.

$8'h8A\}, \{in_2 == 8'h01\}\}$ is satisfiable. In this way, we can find all valid alternative blocks, which are BB2, BB3, BB4 and BB5 in both clock 2 and clock 3. The next step is to sort these blocks by the distance and clock cycle. Since BB3 and BB4 have smaller distance than BB2 and BB5, one of the possible order is $\{\text{BB3}^2, \text{BB4}^2, \text{BB3}^3, \text{BB4}^3, \text{BB2}^2, \text{BB5}^2, \text{BB2}^3, \text{BB5}^3\}$, where the order of BB3 versus BB4 and the order of BB2 versus BB5 in the same clock are random since each pair has the same distance. Assume that we select BB3 in clock 2 as our best alternative block in line 11. A new constraints vector will be constructed, consisting of all constraints from the beginning to BB3 in clock 2. We solve the constraints to get a test and simulate it. Let us assume that the new path is P2. Before searching in the next iteration, we set $clock = 2$ such that only the sub-path after clock 2 is checked in searching for valid alternative branches for P2, i.e., the sub-path (first two clocks) of P2 is locked. A new iteration will repeat the process until the target is activated. There are two key ideas in our algorithm.

1. The usage of our distance metric defined in Section 4.2.3 provides our greedy algorithm a heuristic to explore relatively "close" paths to the targets. For example, in our first iteration, we would prefer P2 over P3 in Figure 4-4 since P2 has smaller distance than P3.

2. The usage of $clock$ maximizes the exploitation of previous good choices, and avoids toggling best alternative blocks when multiple blocks have the same distance. If the $clock$ is not enforced in our algorithm, the best alternative blocks would toggle between BB3^2 and BB4^2 in the following iterations of the previous example. If the explored path could not cover the target, which is likely since we allow the input signal to be random in the remaining cycles, the process will continue until one lucky test activates the target by chance.

4.2.4.1 Dynamic Distance Update

One important thing in our algorithm is the usage of $clock$. The intuition behind $clock$ is that since we have made so much effort in finding the best alternative branch (block), we do not want it to be replaced until we have tried enough possibilities and could not find a solution. Whenever we find an alternative block in line 11, we set $clock$ to the clock cycle of the chosen block. Therefore, the sub-path from the beginning to the chosen block is "locked"

in the following iterations. The next alternative branch is chosen from the remaining cycles. After $clock$ reaches the unrolled cycle, we reset the $clock$ and increment the distance of current path p in line 21-23.

The intuition behind dynamic distance update is that we cannot fully rely on the distance given by our static analysis of CFG. Since the distance is statically computed, it is likely that all paths through a block with a small distance could not activate our target, which is almost impossible to examine because the number of possible paths is exponential. Therefore, we stick with the reasonable evaluation (block-level satisfiability) and apply dynamic distance update to mitigate the shortsighted nature of our edge realignment scheme.

In Algorithm 2, distance is updated when $clock$ reaches unrolled cycles and the target is not covered (line 21). In previous iterations, we have greedily chosen a few profitable alternative branches. We increment the distance of all blocks in the current path p, which contains all the chosen blocks in previous iterations. Through dynamic distance update, our approach is able to explore other blocks instead of exploiting the good blocks from static realignment all the time. For example, assume the distance of BB2 is 10, rather than ∞ in Figure 4-4. After 10 times of trying BB3 and failing to activate the target, the distance of BB2 would be smaller than BB3. Therefore, in the next iteration, BB2 will have higher priority of being chosen as the alternative branch than BB3. Combining the usage of $clock$ and dynamic distance update, our approach balances the exploration and exploitation during path selection.

4.3 Optimizations for Covering Multiple Targets

To extend our test generation framework to handle multiple targets, we propose two optimization techniques: target pruning and target clustering. The focus of these techniques is to effectively utilize the structure of the design and previous search information to generate tests efficiently as shown in Figure 4-1(d). The key idea behind these two techniques are summarized below:

1. Target pruning is designed to reduce the number of targets without sacrificing the coverage. While existing target pruning techniques try to remove redundant targets after

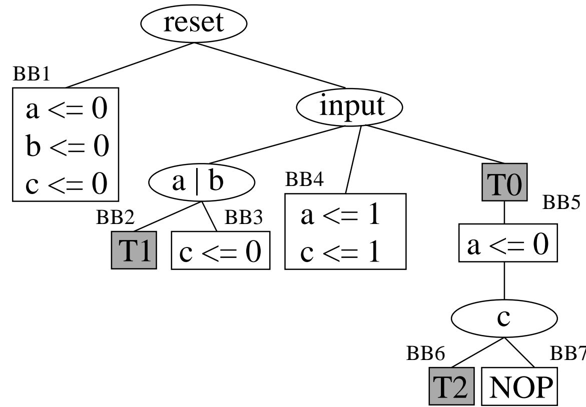


Figure 4-5. CFG for the design in Listing 4.2. T0, T1, and T2 represent three targets.

generating test for all of them, we utilize CFGs and the order of targets to efficiently prune targets prior to and during test generation.

2. Proposed target clustering connects each target with the closest simulated path, therefore, improves initial path selection. One problem of iteratively applying Algorithm 2 is that it ignores all precious search information while activating previous targets, leading to wasted effort in overlapping searches (see Figure 4-1(c)). Target clustering dynamically groups the remaining targets into different clusters. Each cluster has its own closest simulated path. When one target is selected as the current target, we use the closest simulated path as the initial path to replace line 4 in Algorithm 2.

4.3.1 Target Pruning

To accelerate the speed of concolic testing in multi-target scenarios, we prune redundant targets by analyzing the CFG and controlling the order of targets. We exploit both static and dynamic pruning to minimize the number of targets. To illustrate the static process using CFG, consider the simple RTL design in Listing 4.2 as an example. Figure 4-5 shows its CFG with T0, T1 and T2 to represent the three targets. If T2 is reachable, we can safely remove T0 from the target list. Formally, we can prune all the dominator nodes of the targets. Suppose the initial set of targets is TS. For each target $T \in TS$, let the dominators of T be the set $DM(T)$. Therefore, the effective target set after pruning, $TS' = TS - \cup_{T \in TS} DM(T)$.

However, this approach may not work when T2 is not reachable, but T0 is reachable. In this case, removing T0 from the target list does not make sense. Rather, we should remove T2. One engineering choice would be to prune targets as usual, but keep track of the pruned

targets. If a test cannot be generated for a target (e.g., in a reasonable time), add back the dominators that were pruned because of this target. To avoid directly pruning targets for both efficiency and coverage, we topologically sort the targets. The order ensures that the target T_d is always behind the target T_o in the target queue, where T_d is a dominator of T_o. This way, test generation for T_d will only be done if it is not covered by previously generated tests for T_o. For targets in a dominator chain, the deep targets in CFG will always be in front of the shallow ones. For examples, if the original target queue is <T0, T1, T2>, it would become <T1, T2, T0> after target pruning.

Listing 4.2. Example 2

```
module top(clock, reset, in, out);
if (reset == 1'b1) begin
    a <= 0; b <= 0; c <= 0;
end
else case (input)
    2'b00:
        if (a | b)  $display("Target T1");
        else c <= 0;
    2'b10, 2'b01: begin
            a <= 1; c <= 1;
        end
    2'b11: begin
        $display("Target T0");
        a <= 0;
        if (c)      $display("Target T2");
        end
endcase
```

Dynamic target pruning also takes advantage of the order of targets to fully utilize previously explored paths. When the explored paths of a target can cover the other targets, the latter can be pruned. However, it is unknown which targets can be pruned in the beginning.

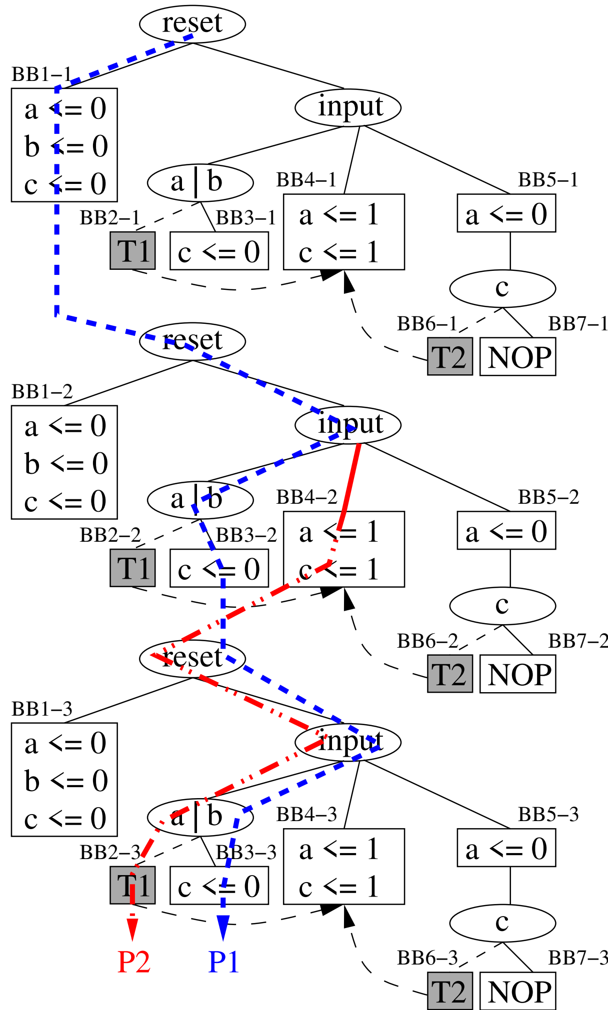


Figure 4-6. Two simulation paths for the design in Listing 4.2 (unrolled for three cycles).

Therefore, we propose a round-robin scheduling in selecting targets. Instead of trying to solve one target until timeout in one round, we split the iteration limit into multiple rounds. If a target cannot be activated in one round, we put it to the end of the target queue. There are two advantages of this scheduling. First, the pruned target may be covered while generating tests for other targets. Second, target clustering may find a better initial path for this target in the following rounds, as introduced in next section.

4.3.2 Target Clustering

Our approach learns target clustering dynamically, and utilizes the clustering to achieve the most profitable initial path for concolic testing. There are mainly two advantages in

selecting a profitable initial path. The first is to improve test generation efficiency. When the initial path is already close to the target, fewer concolic iterations are needed to activate the target compared to initial paths that are far away. The second advantage is to improve coverage. Although coverage is mainly controlled by how an alternative branch is selected, a better initial path means fewer concolic iterations, reducing the probability of getting lost in a large number of misleading alternative branches.

Since current designs separate different functionalities into independent modules, one random simulation path may be far away from our desired target (e.g., if it involves interaction of multiple modules). On the other hand, many targets from the same module or the same finite state machine may share a common path. For these targets, search paths for one target may be close to the other targets. To better utilize the effort of previous explorations, we propose a dynamic clustering approach to learn the most profitable initial path. For each target, we keep the simulated path with the smallest distance evaluated based on the CFGs after edge realignment, called the *closest simulated path*. We place targets in one cluster if they share a common closest simulated path. Initially, all targets are in the same cluster with the closest simulated path being a random path. The simulated path in concolic iteration is used to split clusters into smaller ones.

We use the example in Figure 4-6 which shows the first two steps of exploring paths for the target T1 in Listing 4.2. Assume that the design is unrolled for 3 cycles and $input$ is $2'b00$ for all clock cycles. Then, the initial path is P1. As BB4 has the smallest distance to T1 and it is reachable in the second cycle of P1, the alternative branch (bold solid line) is taken and an input vector is returned by the constraint solver. Assume P2 is the simulated path of the returned input vector. At the same time, targets are dynamically clustered as follows. T1 and T2 are initially in the same cluster with the closest simulated path being P1. After one concolic iteration, P2 is found and used to update the cluster. As P2 visited BB4 in the second clock cycle, it is closer to T2 than P1. Then the cluster and the closest simulated path is updated for T2. When T2 is selected as the current target, we want to start with its closest

simulated path (P2) to avoid overlapping search. This technique effectively eliminates the overlapping search problem. Target clustering also emphasizes the importance of a good edge realignment and distance evaluation scheme. With an incorrect distance evaluation, a target may start from a path that is worse in activating the target, resulting in longer test generation time or failure to activate the target.

4.4 Experiments

4.4.1 Experimental Setup

To evaluate the effectiveness and efficiency of our approach, we compared the performance of our proposed approach with state-of-the-art techniques including uniform test generation (QUEBS) [2] and bounded model checking (EBMC) [4, 100]. The experiments were conducted in a server machine with Intel Xeon CPU E5-2698 @2.20GHz. Our approaches utilize the Icarus Verilog Target API [101] for parsing and generation of abstract syntax tree of RTL code. Prior to applying the framework, the design is first flattened using *flattenverilog* tool from *Design Player Toolchain* [102]. Yices SMT solver is used for constraint solving [103].

4.4.2 Performance Comparison

In this experiment, we compared the performance of our approach to EBMC [4, 100] and QUEBS [2]. A variety of benchmarks are selected from ITC99 [104], TrustHub [105], and OpenCores [96] as shown in Table 4-2. We omit or1200 in the names of the benchmarks or1200_ICache, or1200_DCache and or1200_Exception from OpenCores [96], and omit AES in the names of AES-T1100 and AES-T2000 from TrustHub [105] for simplicity. All these benchmarks contain hard-to-activate branch targets, providing a reasonable test generation complexity. For target selection, we first ran the benchmarks with one million random tests. Then, we selected 20 rarest branches as our targets. For each Trojan-inserted benchmark form TrustHub (AES-T1100, AES-T2000), the selected targets contain 5 rare branches from the Trojan area. There is one rare branch from AES-T2000 that is not included, as it can only be covered after 2^{127} clock cycles. The number of unrolled cycles are chosen such that the hard-to-activate branches can be covered. In practice, a designer can start with a

reasonable number of unroll cycles, and increment it in an iterative fashion until all the targets are covered. The number of unroll cycle problem is the same as the bound determination in bounded model checking [4]. Therefore, after we decided the number of unrolled cycles, we set the same number for the bound in EBMC. We set a new target for EBMC each time, and report the accumulated performance. Since the goal of QUEBS [2] is to cover all branches, we terminated it once it covered all of our selected targets. For the round-robin scheduling of selecting targets in our approach, we set the iteration limit to be 20 in each round.

Table 4-2. Comparison of target coverage using [2], [3] and our approach on 20 targets.

			EBMC [100]			QUEBS [2]			Our Approach			Impro. / EBMC		Impro. / QUEBS	
Bench	cycle	lines	cvr	time	mem	cvr	time	mem	cvr	time	mem	time	mem	time	mem
b10	30	182	20	4.1s	31MB	20	0.12s	9MB	20	0.02s	9.5MB	205x	3.3x	6x	-1.1x
b14	50	698	20	243s	467MB	20	21.6s	34MB	20	1.3s	15MB	187x	31x	17x	2.3x
ICache	50	258	20	6.3s	48MB	20	4371s	1.6GB	20	0.15s	18MB	42x	2.7x	29140x	89x
DCache	10	562	20	20s	138MB	20	1.27s	13MB	20	0.34s	16MB	59x	8.6x	3.7x	-1.2x
Exception	15	666	20	6.9s	40MB	20	3.3s	15MB	20	2.2s	23MB	3.1x	1.7x	1.5	-1.5x
usb_phy	20	1039	20	3.1s	26MB	12	8.2s	34MB	20	134s	138MB	-50x	-5.3x	-16x	-4.1x
T1100	10	544k	20	2386s	8.1GB	-	-	-	20	55s	1.2GB	43x	6.8x	-	-
T2000	10	456k	†	†	†	-	-	-	20	74s	1.2GB	-	-	-	-
Average*	-	-	20	381s	1264MB	19	734s	284MB	20	33s	327MB	69x	7x	4859x	13.9x

†EBMC produced errors and did not finish this benchmark.

*During average computation, we omitted the benchmarks that did not finish.

The performance comparison is shown in Table 4-2. The second column shows the number of unrolled cycles for each benchmark and the third column represents the number of lines of code in each flattened design. For each approach, we report the number of covered targets (*cvr*), the test generation time (*time*) and memory usage (*mem*). All 20 targets are covered in three approaches except that QUEBS only covers 12 branch targets in usb_phy. Although the main idea of QUEBS is to uniformly cover all branches using BFS or DFS, it fails to cover some branch targets due to the trade-offs made by the authors [2] to balance repeated search and overall coverage. Compared to our approach, QUEBS performed worst in two of the benchmarks - b14 and ICache. For ICache, our approach gains 29140 times improvement in test generation time and 89 times improvement in memory usage. This is because these two benchmarks are unrolled 50 cycles. If the number of unrolled cycles keeps growing, QUEBS is expected to face path explosion problem since the total number of branches grows exponentially with the number of unrolled cycles. The scalability issue of QUEBS becomes worse with the complexity of designs. For two Trojan-inserted AES designs (T1100 and T2000) with around 500k lines of code after flattening, QUEBS cannot finish within the time limit (one week). Compared to EBMC, our approach is both time efficient and memory efficient. Note that EBMC reported some errors in AES-T2000, hence the comparison does not include AES-T2000. For the largest benchmark, AES-T1100, our approach can activate 20 targets in 55 seconds with 1.2GB memory usage, while EBMC takes around 40 minutes to finish and consumes 8.1GB memory. For larger benchmarks, we will discuss in Section 4.4.3 to explore the scalability of these two approaches. Overall, our approach provides significant improvement compared to EBMC in both test generation time (69x on average, up to 205x) and memory usage (7x on average, up to 31x improvement).

4.4.3 Scalability Comparison

In this experiment, we examined the scalability of our approach and EBMC. In particular, we compared the memory requirement of our approach to EBMC, since the main challenge of applying model checking to large benchmarks is the state explosion problem. As QUEBS

faces path explosion problem for benchmarks larger than AES, we omitted QUEBS in this experiment and only compared our approach to EBMC. In other words, QUEBS is less scalable than EBMC due to path explosion problem in covering branch targets. Note that the memory requirement is also dependent on the complexity of the design and the branch target. For example, b14 and or1200_Exception have similar number of lines of code. However, the memory requirements of EMBC vary significantly as shown in Table 4-2, with 467MB for b14 and 40MB for or1200_Exception. Therefore, to minimize the impact of other factors and focus on the size of the benchmarks, we used six custom AES benchmarks as shown in Table 4-3. These benchmarks differ only on the number of rounds, which are indicated in the names. For example, aes_20 has 20 rounds compared to 10 rounds in a typical 128-bit AES. By changing the number of rounds, the size of these benchmarks are easily controlled to demonstrate the scalability. The number of lines of code and the total number of branches are shown in the third and fourth columns, respectively. In each benchmark, we inserted one Trojan in the same way as AES-T1100, and created a branch to check if the Trojan is activated. This branch is selected as the target and it is not covered by millions of random simulations. The number of unrolled cycles are five cycles more than the number of rounds to ensure that the branches in all rounds have a chance to be activated.

The experimental results are shown in Table 4-3. As we can see, the average memory reduction of our approach compared to EBMC is 10 times. For the largest benchmark aes_40 with 1.7 million lines of code after flattening, EBMC needs at least 34GB memory while our

Table 4-3. Comparison of memory requirement using EBMC and our approach on one target.

Bench	cycles	lines	total branches	Memory Requirements (GB)		
				EBMC	Our	Reduction
aes_15	20	544k	123k	6.4	0.9	7.1x
aes_20	25	668k	164k	10.3	1.3	7.9x
aes_25	30	886k	205k	15.0	1.6	9.4x
aes_30	35	1003k	246k	20.7	2.1	9.9x
aes_35	40	1169k	287k	27.1	2.5	10.8x
aes_40	45	1693k	328k	34.3	3.0	11.4x
Average	-	994k	225k	19	1.9	10x

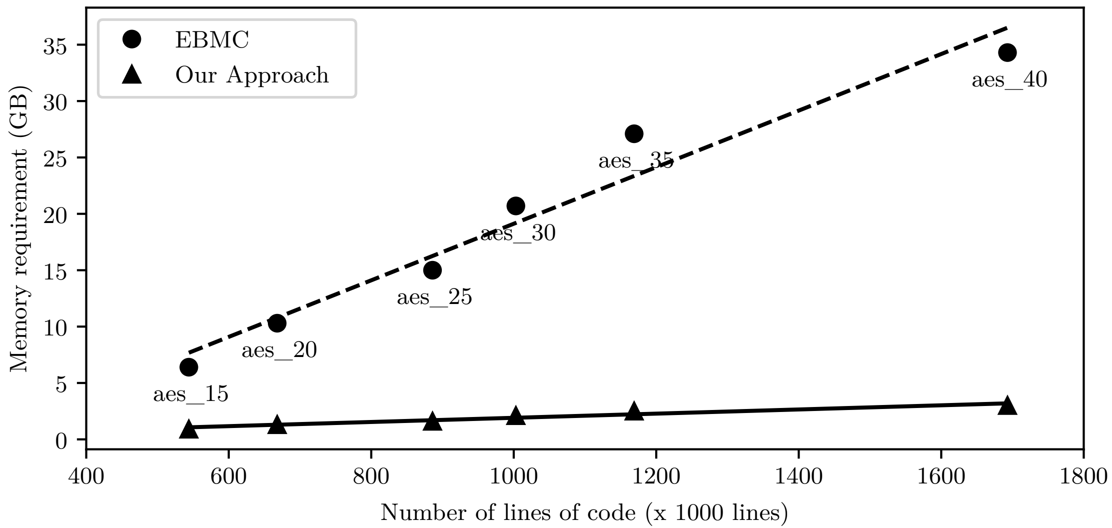


Figure 4-7. The comparison of memory requirements of our approach and EBMC.

approach only requires 3GB. Another observation is that the reduction of memory requirements grows with the size of the benchmarks. For the smallest benchmark, our memory reduction is 7.1 times, and it goes up to 11.4 times for the largest benchmark. The trend of memory requirements can also be viewed from Figure 4-7. The x-axis represents the number of lines of code after flattening, and the y-axis represents the minimum memory requirement. As we can see, EBMC has a much steeper slope than our approach due to its state explosion problem. It is expected that when the size of the benchmarks keep growing, EBMC will give up running much faster than our approach. In other words, our approach is more scalable compared to state-of-the-art model checking tools as well as concolic testing approaches in RTL models.

4.4.4 Effect of Target Pruning

Due to target pruning, some targets are covered by the explored paths for the other targets. The number of pruned targets are shown in Figure 4-8. We also compared the number of targets that can be pruned by EBMC. If a target is already activated by a test that is generated by EBMC to cover a previous target, this target is omitted and counted as a pruned target. Therefore, the number of pruned targets by EBMC is partially affected by the order of targets. For a fair comparison, we fed the targets to EBMC in the same order as our approach.

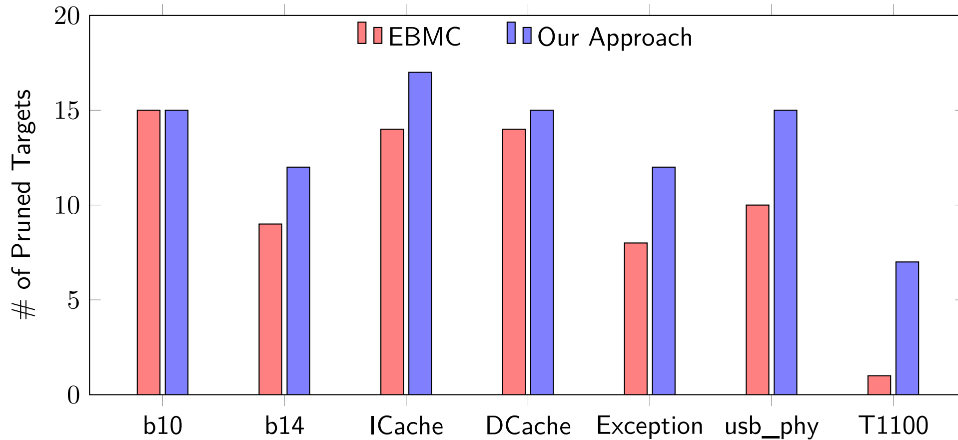


Figure 4-8. The number of targets that are pruned.

As shown in Figure 4-8, both our approach and EBMC pruned over half of the targets for most of the benchmarks. However, our approach achieves consistently better results compared to EBMC. In particular, for the small benchmarks, such as b10 and ICache, the number of pruned targets are similar. On the other hand, the gap of pruned targets is becoming larger when the benchmarks become larger. There are two primary reasons for the success of our approach. First, EBMC is used as a directed test generation scheme. The generated test can cover the branch targets that reside in the same simulation path of the test. As a result, our approach is highly likely to cover these branch targets as well by our simulation. Second, our approach explores many paths from the initial path to our final path to activate one specific target. These paths may come from different parts of the design, and therefore, it is likely to cover other targets (target pruning) or be close to some future targets (target clustering).

For small benchmarks, the hard-to-activate branches are prone to reside in the same rare area. Therefore, EBMC performed well in small benchmarks. However, the hard-to-activate branches in large benchmarks are scattered in different parts of the design. The directed tests generated by EBMC pruned less targets in this scenario. On the other hand, targets are possible to be covered by some paths when our approach is searching for solutions for previous targets. The number of pruned targets reflects the effectiveness of target pruning. It also demonstrates that the extremely hard-to-activate targets dominate the overall test generation time.

4.4.5 Effect of Edge Realignment

To demonstrate the contribution of an efficient edge realignment in our framework compared to a naive edge realignment [3], we applied our approach on the example shown in Listing 4.1 with BB7 as our target. We profiled the number of times each block is chosen as the best alternative block through all iterations in Table 4-4. Assume that the number of selections of BB2, BB3, BB4 and BB5 are x_2, x_3, x_4 and x_5, respectively. The target is activated only when $x_3 + x_4 - x_2 - x_5 > 10$ by statically analyzing the code in Listing 4.1. The first row shows the selection of [3], where the first four blocks are selected almost randomly, as expected from the realignment results shown in Figure 4-3(b). With this random selection, the target is not covered. On the other hand, our approach activated the target in 11 iterations with BB3 and BB4 being selected by 6 and 5 times, respectively.

4.5 Summary

Test generation is an important step during validation and debugging of hardware designs. Conventional validation methodology using random and constrained-random tests can lead to unacceptable functional coverage under tight deadlines. While application of concolic testing on hardware designs has shown some promising results in improving the overall coverage, they are not designed for covering specific targets such as uncovered corner cases and rare functional scenarios. In this chapter, we proposed a scalable test generation framework using concolic testing to automatically activate targets in RTL models. This chapter made two important contributions. (1) We proposed a directed test generation framework in activating a single target utilizing contribution-aware edge realignment and effective path exploration. (2) We developed two optimization techniques to drastically reduce the overall test generation effort involving multiple targets: (i) target pruning to remove the targets that can be covered

Table 4-4. The number of iterations that each block is selected as the best alternative block in exploring paths for Listing 4.1.

Blocks	BB2	BB3	BB4	BB5	BB6	Cover
[3]	20	19	18	18	5	No
Our approach	0	6	5	0	0	Yes

by the tests generated for other targets, and (ii) target clustering to minimize the overlapping searches by utilizing learning from previous searches. Experimental results demonstrated that our approach is significantly faster compared to state-of-the-art test generation techniques. Compared to QUEBS, our approach provides significant speedup in test generation time (up to 29140X, 4859X on average). Similarly, compared to EBMC, our approach provides drastic improvement in test generation time (up to 205X, 69X on average) and an order-of-magnitude reduction in memory requirement.

CHAPTER 5
TEST GENERATION FOR ACTIVATION OF ASSERTIONS

One major challenge in assertion-based validation is to efficiently activate all assertions. Coverage of all assertions is fundamentally different from code coverage due to the vacuity problem. For example, in the formula $p \longrightarrow q$, it is vacuously valid if p is always false. Assertion coverage requires p to be true and q to be false when executing this line, while coverage coverage does not impose this requirement. In practice, designers need to manually write directed test patterns to cover many hard-to-activate assertions. As expected, manual test writing can be time consuming and error prone (requiring numerous trials and errors) - may not be feasible for large designs. Directed tests are promising in activating assertions since a significantly smaller number of directed tests can achieve the same coverage goal compared to random or pseudo-random tests. While existing test generation using model checking is promising, it cannot generate directed tests for large designs due to state space explosion. Simulation-based verification can handle large designs but cannot guarantee activating the assertions due to exponential input space complexity (test patterns). In transaction-level, Ferro et al. [106] proposed a framework for supervising SystemC TLM simulation of PSL temporal properties. Combinatorial testing tools are used to generate test set, which contains all combinations of the values identified by the test engineer. In register-transfer level (RTL), Pal et al. [107] assumed that assertions are defined over the interface of a module (input and output) and proposed an approach to bias random test generation for assertion coverage.
In this chapter, I propose a test generation framework to activate SoC security assertions that does not impose any restriction on assertions variables, and enables test generation for activating security assertions by converting assertions to equivalent branches and activating them utilizing concolic testing. While early work of applying concolic testing to activate functional on RTL models is promising, there are no prior efforts in activating RTL assertions using concolic testing.

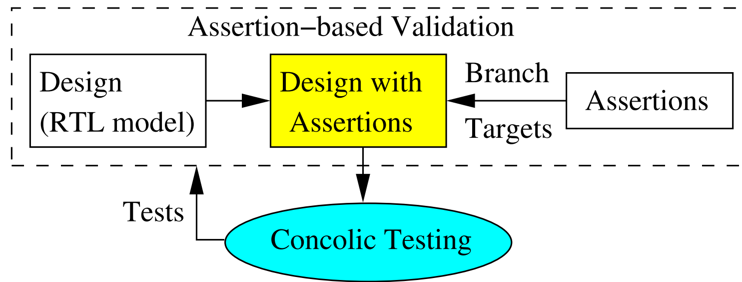


Figure 5-1. Our approach converts assertions to branch targets and activates them non-vacuously.

Our proposed methodology consists of two major steps as shown in Figure 5-1. The first step converts these assertions to branch statements and embed them into the design. Then, it utilizes concolic testing to generate a compact test set to efficiently cover (activate) the target branches (assertions), as illustrated in Chapter 4. While formal methods try to explore all possible paths at the same time (can lead to state space explosion), concolic testing has the inherent advantage of scalability since it explores one execution path at a time. Note that the embedded branch targets are used for test generation purpose only. Once test generation is completed, these branch targets should be removed from the design (RTL model) and replaced with the original assertions. This chapter makes two important contributions:

1. We map the problem of activating assertions non-vacuously to the problem of concolic testing by converting assertions to branch targets (Section 5.2).
2. We propose an efficient test generation method using concolic testing to cover the generated branch targets. The generated test vectors are guaranteed to activate the corresponding assertions (Section 5.3.1).
3. To address the path explosion problem in concolic testing, we efficiently select the most profitable branches to quickly reach the target (Section 5.3.2).

The remainder of the chapter is organized as follows. We present the problem formulation in Section 5.1. Section 5.2 describes the conversion from assertions to branch targets. Section 5.3 presents our test generation framework using concolic testing to activate the branch targets (assertions). Section 5.4 presents experimental results. Finally, Section 5.5 concludes the chapter.

5.1 Problem Formulation

In this chapter, **activation of assertions refers to finding counter-examples that fails the assertions non-vacuously**. Vacuity is defined in [108] as follows: if there exist a sub-formula ψ of a formula ϕ such that ψ can be replaced with arbitrary formula and does not affect the outcome of model checking, the formula ϕ is vacuous in model M. For example, in the formula $p \longrightarrow q$, it is vacuously valid if p is always false, since we can replace q with any sub-formula. We address the vacuity problem by converting the formulas into specific branch targets and applying concolic testing to activate them.

Listing 5.1 shows the branches that are converted from two types of assertions (immediate assertions and concurrent assertions) in Arbiter. Note that the conversions from assertions to branches are the same for these two types, except that an individual concurrently running block is needed to wrap the branches from concurrent assertions. In Listing 5.1, the first assertion is an immediate assertion and its corresponding branch is directly embedded in the same place as the assertion. On the other hand, the second assertion is converted into an always block that is running concurrently with all the other blocks. To find counter-examples that make the assertions fail non-vacuously, we need to generate tests to activate branch targets that are converted from the assertions.

Listing 5.1. An example of branch conversion in Arbiter

```
module arb(clk, rst, req1, req2, gnt1, gnt2);
input clk, rst, req1, req2;
output gnt1, gnt2;
reg state, gnt1, gnt2;
always @ (posedge clk or posedge rst)
    if (rst)
        state <= 0;
    else
        state <= gnt1;
always @ (*)
    if (state) begin
        gnt1 = req1 & ~req2;
        gnt2 = req2;
        // Assert 1: assert(req2 == gnt2)
        if (req2 != gnt2)
            // target 1
    end
    else begin
        gnt1 = req1;
        gnt2 = req2 & ~req1;
    end
// Assert 2: assert property(gnt1|->~gnt2)
always @ (*)
    if (gnt1)
        if (gnt2)
            // target 2
endmodule
```

5.2 Conversion of Assertions to Branches

To generate a test to activate assertion P, we first map the assertion activation problem to branch coverage problem in concolic testing. Algorithm 3 shows our procedure to convert

assertion P to blocks containing a corresponding branch target. Section 5.3 will demonstrate how to use concolic testing to generate tests to cover branch targets. In this section, we introduce the details of converting assertions to branches. In this chapter, we consider assertions with logic operator, implication ($|->$) and delay ($##$). Other operations are not described due to space limitation, but can be converted to branches in similar ways.

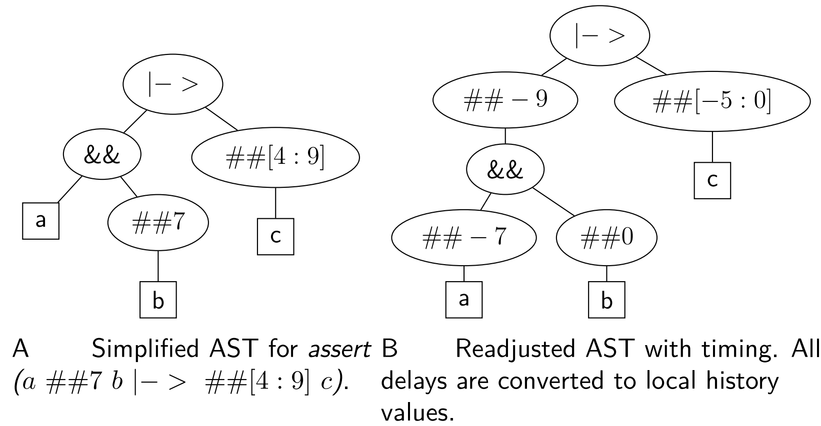
Algorithm 3 Assert to Branch conversion

1: **procedure** ASSERT2BRANCH(assertion P)
2: Construct simplified AST for assertion P
3: Readjust AST with delay information
4: Empty stack S
5: **for** Post-order traversal readjusted AST **do**
6: **if** current node n is an implication **then**
7: Convert implication to logic operator
8: **end if**
9: **if** current node n is a variable **then**
10: Push n to S
11: **end if**
12: **if** current node n is delay **then**
13: Pop variable a from S
14: ***Add delay to*** a
15: Push the modified variable to S
16: **end if**
17: **if** current node n is a logic operator **then**
18: Pop all variables of its children from S
19: ***Combine the children with its operator***
20: Push the result to S
21: **end if**
22: **end for**
23: ***Create branch to test the variable in*** S
24: **return** B containing generated blocks
25: **end procedure**

5.2.1 Simplified Abstract Syntax Tree

To understand the meaning of one assertion, we parse the assertion and build an abstract syntax tree (AST) for it. Three types of operators are selected as non-terminal for our simplified AST, i.e., logic operator, implication and delay. Others are treated as terminals. For example, if the original assertions is *assert* (a $##7$ b $|->$ $##[4:9]$ c), which means if a is 1

in clock 0 and b is 1 in clock 7, then c must be 1 in any clock between clock 11 and clock 16. The simplified AST for this assertion is shown in Figure 5-2A.



A Simplified AST for *assert* (a ##7 b |− > ##[4 : 9] c).

B Readjusted AST with timing. All delays are converted to local history values.

Figure 5-2. AST adjustment with timing. Logic operator, implication and delay are non-terminal nodes (oval), and others are terminals (rectangle).

5.2.2 Adjust AST with Timing

As delays represent the future events, which cannot be evaluated in the current clock cycle, we transform delays into retrieving history values. We assume that there exists a global clock counter (as shown in Listing 5.2), and the design remembers all the "necessary" history values. We use a[clk_cnt] to represent the history value of a in clock clk_cnt. Figure 5-2B shows the readjusted AST for Figure 5-2A. There are two things to consider:

1. Adjustment is local to its own children for each non-terminal nodes. For example, the left sub-tree in Figure 5-2A (a ##7 b) adjusts the delay of 7 to its left child. If we look at the whole expression, the history values of a should be at least 11 cycles ahead of c. This localization property make adjustment efficient.

2. For delay range, we adjust the longest delay to the left side and modify the range appropriately, e.g., $\#\#[4 : 9]$ in Figure 5-2A rotates the $\#\# - 9$ to the left side and adjusts itself to $\#\#[-5 : 0]$.

Listing 5.2. Global clock counter

```
always @(posedge clock) begin
    clk_cnt <= clk_cnt + 1;
end
```

5.2.3 Conversion of AST to Branch Target

After we adjust AST with timing, each node is attached with non-positive delay (implicitly 0 delay). From adjusted AST, we construct branches by post-order traversal of the adjusted AST with the help of a stack S. Each part of the clause is represented by a unique variable except for the clauses which can be directly accessed. Stack S contains the visited variables that have not been combined by other clauses. Algorithm 3 shows how the target branch is generated (in italic bold text) with the help of stack S. The generated code of Figure 5-2B is shown in Listing 5.3. As shown in Algorithm 3, RTL code is generated based on the root type of each sub-tree. We consider the following three root types:

Listing 5.3. The branch converted from Figure 5-2B

```
always @(posedge clock)
begin
    // p1 = ##-7 a && b
    p1[clk_cnt] = a[clk_cnt - 7] && b[clk_cnt];
    // p2 = ##[-5:0]c
    p2[clk_cnt] = 0;
    for (i : [clk_cnt - 5, clk_cnt])
        p2[clk_cnt] = p2[clk_cnt] | c[i];
    // p3 = ##-9 p1 |-> p2
    p3[clk_cnt] = 1;
    if (p1[clk_cnt - 9])
        if (!p2[clk_cnt])
            p3[clk_cnt] = 0;
    // branch target
    if (!p3[clk_cnt])
        $display("Assertion fail");
end
```

Delay: For a single delay, we retrieve the history value of the variable, e.g., when we visit the node $\#\# - 7$ in Figure 5-2B, the node a is in the top of stack S. We pop a from S, and push back a[clk_cnt - 7]. A delay range represents an OR operation on all the values, e.g.,

$\#\#[-5:0]c$ means $c[-5]|c[-4]|...|c[0]$. Listing 5.3 shows the expansion of $\#\#[-5:0]c$ using for-loop and uses variable $p2$ to represent this part. When a single delay is applied, we skip generating a new variable for the clause, e.g., $\#\#-7a$ directly utilizes the history value of a instead of generating a new variable.

Logic Operator: When the root is a logic operator, Algorithm 3 combines all its children (contains delay information) using the operator. As each child is already represented by a single variable in the stack S, we just pop all of them from S, and use a new variable to represent the combined result.

Implication: Implication, A $|->$ B, contains two parts: A is called the antecedent, and B is called the consequent. There are two implication operators in SVA, i.e., overlapped implication ($|->$) tests consequent sequence at the clock when its antecedent sequence is activated, while nonoverlapped implication ($|=>$) tests the consequent in the next clock cycle. The latter one can be converted to the previous one by adding one cycle delay to the consequent sequence. As shown in Listing 5.3, we convert the implication node into variable $p3$.

When we finish traversing the readjusted AST, the assertion expression is represented as a single variable in top of stack S, e.g., $p3$ in Listing 5.3. A branch target is created by checking the value of the final variable.

5.2.4 Complexity Analysis

For the ease of representation, we assumed that the design remembers all "necessary" values in the previous iterations. To achieve memory efficiency, the clk_cnt can be as small as the largest delay in the whole assertion, e.g., 9 for *assert($a\ \#\#7\ b\ |->\ \#\#[4:9]\ c$)*, as a result of introducing new variables. If we look at the code in Listing 5.3, the impact of a[clk_cnt - 16] is already stored in p1[clk_cnt - 9]. Thus, remembering older values than the longest delay is a waste of memory. After determining the largest delay, we add a module operation to Listing 5.2, i.e., clk_cnt<= clk_cnt $\mod$ (9 + 1), with an extra one to remember the current clock. Assume that b is the longest delay and n is the length of the

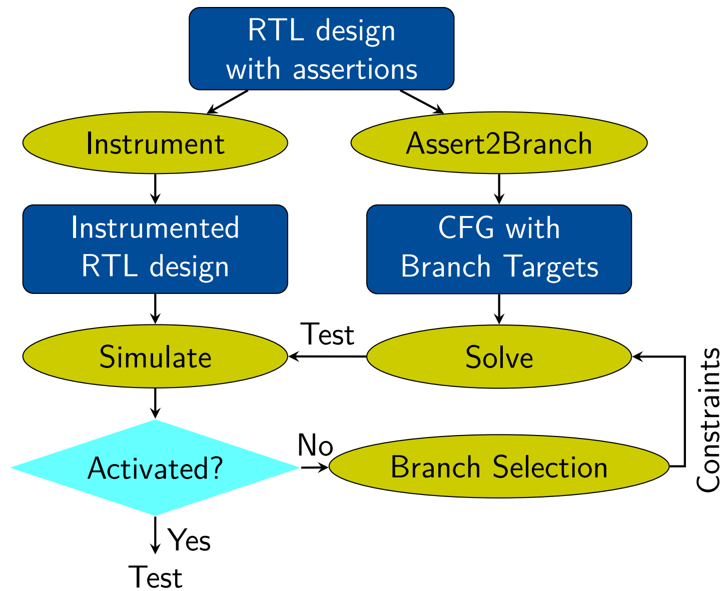


Figure 5-3. Overview of our framework to activate the branches converted from assertions.

assertion. The *memory requirement* complexity is $O(bn)$ since the memory usage of the tree structure, the stack S, and required new variables are linear to the length of assertion. The running time of Algorithm 3 is dominated by post-order traversal of the AST, compared to the AST construction and adjustment. For each node, the running time is linear to the number of children. Then, each node contributes twice to the total running time. Since the number of nodes in AST is linear to the length of the assertion, the *running time* complexity is $O(n)$.

5.3 Test Generation using Concolic Testing

Once the assertions are converted to branches, we apply concolic testing to generate tests to cover the generated branch targets. This section is organized as follows. First, we provide an overview of our test generation framework. Next, we briefly discuss efficient selection of alternate branches.

5.3.1 Overview

Figure 5-3 shows the overview of our test generation framework. Concolic testing combines concrete simulation and symbolic execution, as shown in Figure 5-3. The left side shows the concrete simulation part, and the right side shows the symbolic execution part. To instruct symbolic execution, the concrete path needs to provide every branch it takes. Instead

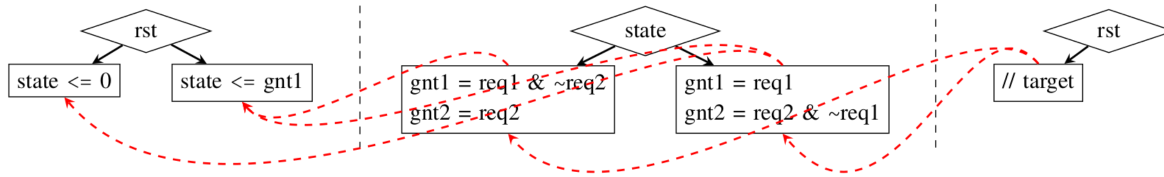


Figure 5-4. Chaining of related blocks in CFGs to assist alternative branch selection.

of changing simulator to execute symbolically in each branch and assignment, we use existing tools for simulation, and instrument the RTL design with *display* statement to show which branch the simulation has taken. For example, the instrumented first block of Listing 5.1 is shown in Listing 5.4.

Based on Algorithm 1, the assertions in RTL design are converted into branch targets in control flow graph (CFG). For every test that is generated by symbolic execution, simulation will give the concrete execution and report every branch it takes. Based on the branch information, constraints are constructed together with all the assignments inside the corresponding blocks. The most important step in concolic testing is to find the best alternative branch to flip, which will be discussed in the next section. With the selected alternative branch, new constraints are constructed, and solved by an SMT solver to generate a new test for simulation. The general idea is to efficiently explore different paths to get closer to the branch target converted from a specific assertion.

5.3.2 Selection of Alternate Branches in CFG

To help alternative branch selection, we first chain the relative blocks together in control flow graph. We use the second assertion in Listing 5.1 as an example. The branch target is controlled by the condition *gnt1* $\&$ *gnt2*. Therefore, the target block is chained to the blocks where either $gnt1$ or $gnt2$ might be assigned '1'. Similarly, since the blocks in the second CFG are controlled by the value of $state$, the blocks are chained to the blocks in the first CFG, as shown in Figure 5-4.

Listing 5.4. Instrumented first block in Arbiter

```
always @ (posedge clk or posedge rst)
    if (rst) begin
        $display("arb2 branch 1 taken");
        state <= 0;
    end
    else begin
        $display("arb2 branch 2 taken");
        state <= gnt1;
    end
```

This chaining process helps alternative branch selection concentrating only on related branches. When we consider the relevance of one branch with the target, we calculate the distance from the immediate block following the alternate branch to the target. In each iteration, the most relevant and reachable branch is selected as the alternative branch to construct new constraints and generate a new test.

5.4 Experiments

This section is organized as follows. First, we describe our experimental setup and an example of inserted assertions. Next, we present our test generation results.

5.4.1 Experimental Setup

To evaluate our test generation technique in activating assertions non-vacuously, we implemented our framework in C++ using Icarus Verilog Target API [109] with Yices [103] as the constraint solver. As shown in Section 5.3, our framework first converted all assertions to branches and inserted them into modified designs. Next, it applied concolic testing to generate test to activate the branches. Finally, we simulated the assertion-inserted instances (before converting to branches) to validate the correctness of generated test sets. Our framework is compared with EBMC [4] to show the performance improvement. All the experiments are performed on a machine with Intel E5-2698 v4 @ 2.20GHz CPU.

5.4.2 Benchmarks and Assertions

We selected 12 benchmarks to evaluate our framework. The first three benchmarks, wb_conmax-T200, AES-T1000 and AES-T1100, are from TrustHub [105]. The remaining benchmarks are custom benchmarks of AES, named as cb_aes_n, where n is the number of rounds in AES. We varied the number of rounds to easily control the size of our benchmarks. To show the inserted assertions, we use AES-T1000 as an example. Listing 5.5 shows the Trojan_Trigger module from AES-T1000 benchmark [105]. The inserted assertion is $assert\ property(rst == 0)|- > (trig == 0)$. In most scenarios, the extremely rare branch in Listing 5.5 is never executed. As a result, traditional testing approaches using millions of random tests will not activate this assertion non-vacuously. For other benchmarks, we inserted assertions in the same way.

Listing 5.5. Trojan_Trigger module in AES-T1000 [105]

```
module Trojan_Trigger (rst, state, trig);
  input rst;
  input [127:0] state;
  output trig;
  always @ (rst, state)
    if (rst == 1'b1) trig <= 1'b0;
    else begin
      if (state==128'h00112233445566778899aabbccddeeff)
        trig <= 1'b1;
  end
endmodule
```

5.4.3 Test Generation Results

In this experiment, we applied our framework to generate tests for assertions. The performance comparison is shown in Table 5-1. The number of unrolled cycles is just enough to activate the assertions. For example, the number of unrolled cycles is $n + 5$ for each custom

Table 5-1. Performance comparison of our approach with EBMC [4] in activating assertions.

	EBMC [4]		Our Approach			
Benchmark	Time (s)	MEM (GB)	Time (s)	Time Imp.	MEM (GB)	MEM Imp.
wb_conmax	5.42	0.74	6.86	-1.3x	0.13	5.7x
AES-T1000	116	7.99	8.91	13x	0.60	13x
AES-T1100	116	7.99	21.43	5.4x	0.69	12x
cb_aes_01	2.89	0.17	0.90	3.2x	0.06	2.9x
cb_aes_10	58.4	3.42	12.81	4.6x	0.59	5.8x
cb_aes_15	113	6.42	27.9	4.1x	0.88	7.3x
cb_aes_20	178	10.3	63.7	2.8x	1.23	8.4x
cb_aes_25	260	15.0	127	2.1x	1.58	9.5x
cb_aes_30	411	20.7	230	1.8x	1.97	11x
cb_aes_35	478	27.1	372	1.3x	2.36	12x
cb_aes_40	617	34.3	578	1.1x	2.81	12x
Average	214	12.2	132	3.5x	1.17	9.1x

AES benchmark cb_aes_n as it requires n cycles to get results from output. As shown in Table 5-1, our approach is significantly faster (up to 5.4x, 3.5x on average) than EBMC.

Our approach is also more efficient in memory usage. As shown in Table 5-1, our approach is up to 13x (9.1x on average) more efficient in memory usage compared to EBMC. To better visualize the relationship between the memory requirement with respect to the size of the design, we plot the memory requirement of two approaches for our custom benchmarks in Figure 5-5. Note that the number of lines for each custom AES benchmark is the total lines after hierarchy flattening. As we can see, the memory requirement of EBMC grows exponentially with the lines of code. It is due to the state space explosion problem of model checking. On the other hand, the memory requirement of our approach grows linearly with the lines of code since it explores one path at a time, which is linear to the code size. For the benchmark cb_aes_40 (around 1.3 million lines of code), EBMC requires over 34GB memory, while our approach only needs 2.8GB. Due to exponential memory requirement, EBMC is expected to fail for larger and more complex designs, while our approach is expected to be scalable since memory requirement increases linearly.

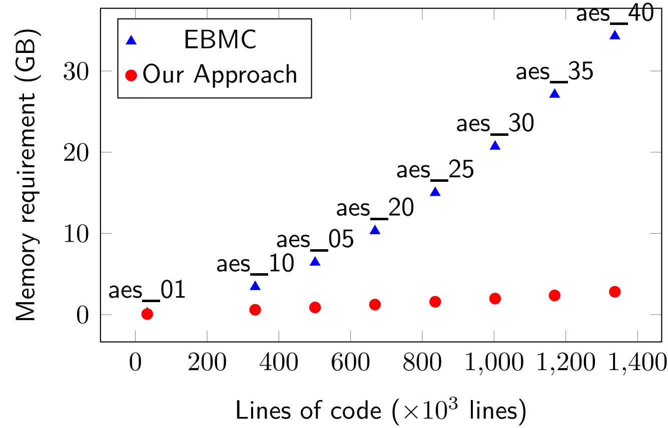


Figure 5-5. Memory requirement with respect to the total lines of code in custom benchmarks.

5.5 Summary

Assertions are widely used in validation of hardware designs (RTL models). A major challenge in assertion-based validation is how to activate all the assertions to ensure that they are valid. While existing model checking based directed test generation is promising, it cannot generate tests for large designs due to state space explosion. We presented an automated and scalable mechanism to generate directed tests using concolic testing to activate assertions non-vacuously. Using a diverse set of benchmarks, our experimental results demonstrated that our test generation approach is significantly faster (up to 5.4x, 3.5x on average) compared to state-of-the-art test generation methods. Most importantly, our approach is scalable since it has linear memory requirement, while state-of-the-art directed test generation method has exponential memory requirement.

CHAPTER 6
TEST GENERATION FOR VALIDATION OF CACHE COHERENCE PROTOCOLS

Computing systems utilize multi-core processors with complex cache coherence protocols to meet the increasing need for performance and energy improvement. It is a major challenge to verify the correctness of a cache coherence protocol since the number of reachable states grows exponentially with the number of cores. Simulation using random and constrained-random tests is widely used in industry because of its good scalability. However, the random nature of test sequences also introduces unacceptable time requirement to cover all possible state transitions in modern cache coherence protocols with many cores. Clearly, it is inefficient to use breadth-first search (BFS) on this product FSM to achieve full state or transition coverage, because a large number of transitions may be unnecessarily repeated, if they are on the shortest path to many other states. Directed tests, on the other hand, are promising to achieve high coverage with a drastically small number of tests [110]. Therefore, they can be applied in addition to random tests to further improve the chances of capturing potential bugs. However, directed test generation is not practical in this case since the time and memory requirements can be prohibitive. Qin and Mishra [35] proposed an on-the-fly test generation technique for cache coherence protocols by analyzing the state space structure of their corresponding global FSMs. Instead of using structure-independent BFS to obtain directed tests, they decomposed complex state space into several components with simple structures. The efficient technique is shown to achieve full state and transition coverage in simulation based verification for a wide variety of cache coherence protocols. While their on-the-fly method can reduce the numbers of required tests by half, it can still be impractical to verify all possible transitions in the presence of large number of cores. In this chapter, we propose scalable on-the-fly test generation techniques using quotient state space. The proposed approach guarantees selection of important transitions by utilizing equivalence classes, and omits only similar transitions. This approach is based on the efficient decomposition technique of graphical state space description in [35] and Euler tour [111] traversal. Specifically, we

propose an efficient quotient space based test generation approach to address the scalability concerns in existing test generation techniques. Quotient space is one of the symmetry reduction techniques. Through defining equivalence classes of states and restricting state space to representatives, the proposed approach enables designers to cover important state transitions within limited verification budget. Our experimental results demonstrate that our proposed approaches can efficiently trade-off between transition coverage and validation effort.

The rest of the chapter is organized as follows. Section 6.1 introduces the background of cache coherence. Section 6.2 briefly introduces the on-the-fly test generation algorithms from [35]. Section 6.3 proposes scalable on-the-fly test generation using quotient space. Experimental results are presented in Section 6.4. Finally, Section 6.5 concludes the chapter.

6.1 Background

In modern computer systems, each processing unit usually maintains its local copy of the main memory, or cache for fast access. One major problem of caching is that when the same data, memory block, is cached in two or more different places, any modification should be propagated to all the cached copies. Cache coherence protocols are used to define the correct behavior of each cache controller.

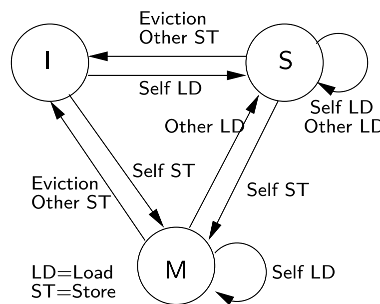


Figure 6-1. State transitions for a cache block in MSI protocol.

One of the simplest cache coherence protocol is the MSI snoopy protocol [112]. The behavior of the cache controller in a processing unit is modeled as an FSM (Figure 6-1). The state of a cache block (line) can be either “Invalid”(I), “Modified”(M), or “Shared”(S). At the beginning, all cache blocks are in the invalid state. When a load request (Self LD) arrives, the

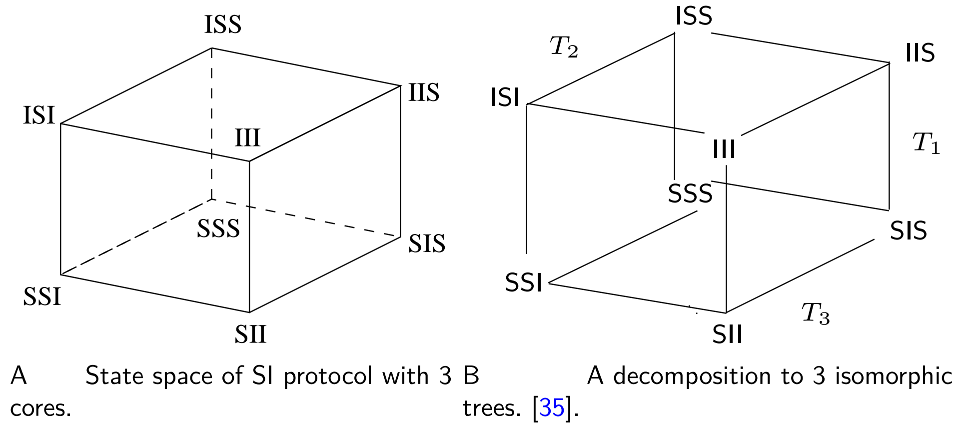
cache controller requests the data from the main memory and switches to shared state. When the core issues a store request (Self ST), the cache controller first broadcasts an invalidate request on the bus and then changes to modified state. Such an invalidate request will inform all the other cache controllers that are in shared or modified states to change to invalid state. A cache block may also change to invalid state, when it is evicted by another cache block, which is mapped to the same location in the cache, or when other cores issue store requests (Other ST).

Although MSI protocol can guarantee the coherence of the cache system, it causes some unnecessary delay and traffic on the communication channels. Researchers have designed many cache coherence protocols for different platforms and architectures, such as MSI, MESI, MOSI, MOESI, MESIF [22], MEUSI [113] and many other variations.

6.2 Test Generation for Validation of Cache Coherence Protocols

As cache coherence protocols are becoming more and more complex, it is getting harder to verify their implementations. From the validation perspective, it is always desirable to activate all possible state transitions of the entire multicore cache system. Directed test generation is promising to provide full coverage of the whole finite state machine. Given the FSM description of any cache coherence protocol, traditional directed test generation is able to compose a test suite which can activate all states and transitions using two steps: 1) for each state, it determines the instruction sequence to reach it by performing a BFS on the global FSM; 2) for each transition, it creates the test by appending the required instructions after the instruction sequence to reach the initial state of this transition. However, such a naive approach has two problems. 1) Transitions close to the initial state are visited many times. Thus, a large portion of the overall test time is wasted. 2) Since it has to remember all visited states in BFS, its runtime memory requirement also grows exponentially. Therefore, traditional breadth-first search is not scalable to large designs due to exponential memory and time requirements. To overcome this problem, Qin and Mishra [35] proposed an on-the-fly test generation algorithms to drastically reduce the test generation time and memory requirements.

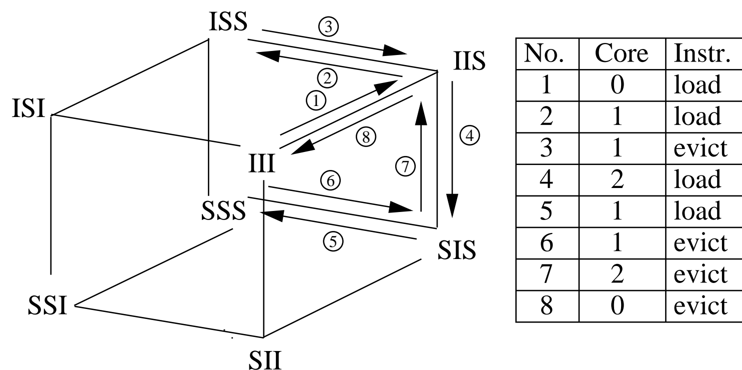
Their approach is based on a global FSM and exploits the highly symmetric and regular structure of the state space and design a deterministic test generation algorithm. Structures like hypercubes and cliques can be traversed by visiting each transition exactly once. For example, after decomposing SI protocol into sub-structures as shown in Figure 6-2, Euler traversing is applied to each sub-structure as shown in Figure 6-3.



A State space of SI protocol with 3 cores.

B A decomposition to 3 isomorphic trees. [35].

Figure 6-2. The decomposition of the state space of SI protocol with 3 cores.



No.	Core	Instr.
1	0	load
2	1	load
3	1	evict
4	2	load
5	1	load
6	1	evict
7	2	evict
8	0	evict

Figure 6-3. The tests generated by Euler traversal of the upper right sub-structure of hypercube.

6.3 Scalable Test Generation using Quotient Space

Since the number of states in coherence protocols grows exponentially as the number of core increases, it may not be realistic to cover all possible transitions of many-core designs within given verification budget. A widely used technique to address this limitation is to perform verification on quotient space. By grouping states into equivalent sets and checking

only the representative state per set, the total validation effort is greatly reduced by eliminating similar transitions. However, since the original state space is prohibitively large to explore, validation on quotient space still faces two critical challenges: 1) how to maximize the utilization of each transition by avoiding revisit of the same transition unnecessarily, and 2) how to make the test simulation/execution time configurable to provide trade-off between state/transition coverage (confidence) and verification budget (available time).

In this section, we are going to address these challenges by extending on-the-fly test generation techniques in [35] to support test generation for many-core coherence protocols using quotient space. For the ease of presentation, we are going to employ several group theory terminology in the following discussion.

Definition 6.1. *Let X be a finite set. A permutation of X is a bijection from X to X. The set of all permutations of X forms a group under composition of mappings. Any subgroup of this group is called a permutation group acting on the set X. We denote permutations using cycle notation. For example, $G_0 = (0, 2)(1, 3, 4)$ acting on $X_0 = \{a_0, a_1, a_2, a_3, a_4\}$ repeatedly performs the following permutation: $a_0 \rightarrow a_2$, $a_2 \rightarrow a_0$, $a_1 \rightarrow a_3$, $a_3 \rightarrow a_4$, $a_4 \rightarrow a_1$.*

Definition 6.2. *Given a permutation group G acting on a finite set X, for $x \in X$ the set $\{\pi(x) : \pi \in G\}$ is called the orbit of x under G, denote $[x]_G$. $G_0 = (0, 2)(1, 3, 4)$ divides X_0 into two orbits: $\{a_0, a_2\}$ and $\{a_1, a_3, a_4\}$.*

Given a set of nodes and a permutation group, we define *orbit state* as follows: the orbit is in

1. I state, if all nodes in the orbit are in I state.
2. S state, if all nodes in the orbit only contain I or S state, and at least one node is in S state.
3. E state, if at least one node in the orbit is in E state.
4. O state, if at least one node in the orbit is in O state.
5. M state, if at least one node in the orbit is in M state.

Let $[\boldsymbol{s}]_G$ be the global orbit state of $\boldsymbol{s}$, where each element of $[\boldsymbol{s}]_G$ is the state of corresponding orbit. We use α to denote the number of orbits.

Definition 6.3. *The quotient protocol P_G of protocol P with respect to permutation group G is a tuple $P_G = (S_G, T_G)$, where $S_G = \{[\boldsymbol{s}]_G : \boldsymbol{s} \in S\}$ (S is the state space of P), $T_G = \{([\boldsymbol{s}]_G, [\boldsymbol{t}]_G) : (\boldsymbol{s}, \boldsymbol{t}) \in T\}$ (T is the transition rule of P). We denote the quotient protocol of certain standard protocol by adding prefix 'P' to the protocol name. For instance, the quotient protocol of MSI is denoted as PMSI.*

Theorem 6.1. *The state space of quotient protocol PSI with n cores and α orbits is equivalent to the state space of an SI protocol with α cores.*

Proof. First, it is easy to see that the global orbit state $[\boldsymbol{s}]_G$ contains α elements, which is the same as the number of elements in the state of SI protocol with α cores. Then, we prove that every state/transition in SI protocol also exists in PSI protocol.

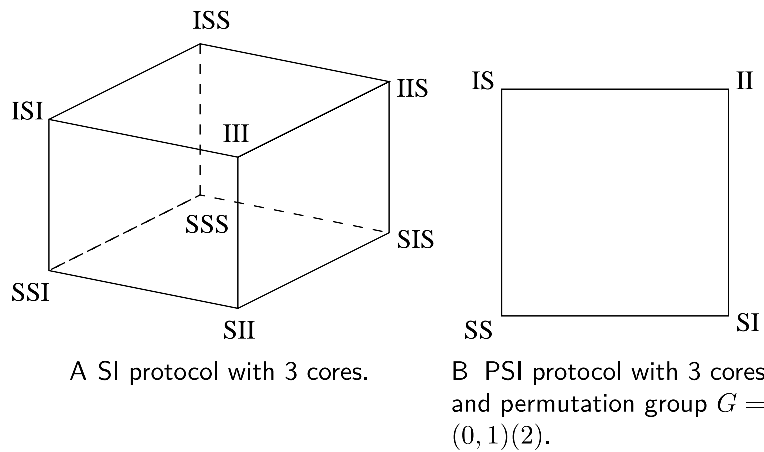
For any state $\boldsymbol{s}$ in the state space of SI protocol, suppose $s_{i_1}, s_{i_2}, ..., s_{i_k} =$ S. To achieve the corresponding state in PSI protocol, we can randomly choose one node from each orbit i_1, i_2, ..., i_k and let their states be S. So, every state in SI protocol exists in PSI protocol.

For any transition from state $\boldsymbol{s}$ to $\boldsymbol{s'}$ in SI, $\boldsymbol{s'}$ either is $\boldsymbol{s}$ itself, or contains one different element. Suppose $s_j =$ S and $s'_j =$ I. To get the corresponding transition in PSI protocol, we first construct the corresponding state of $\boldsymbol{s}$ by the above method. As the above method makes at most one node in each orbit to be in S state, suppose node t in orbit j is in S state. A transition $\boldsymbol{s}$ to $\boldsymbol{s'}$ is achieved by the evict operation of node t. Similarly, we can get corresponding transition if $s_j =$ I and $s'_j =$ S. So, every transition in SI protocol exists in PSI protocol.

We can follow the similar arguments to prove that every state/transition in PSI protocol also exists in SI protocol. Therefore, the state space of quotient protocol PSI with n cores and α orbits is equivalent to the state space of an SI protocol with α cores. □

Example 1: Consider PSI protocol with 3 cores and permutation group $G = (0,1)(2)$. {II} in PSI represents {III} in SI, {IS} in PSI represents {IIS, ISI, ISS} in SI, {SI} in PSI

represents {SII} in SI, and {SS} in PSI represents {SIS, SSI, SSS} in SI. Figure 6-4a shows the state space of the original SI protocol with 3 cores and Figure 6-4b shows the corresponding PSI protocol. It is easy to verify that the state space of PSI is equivalent to that of SI protocol with 2 cores. Every transition in PSI is a Cartesian product of the respective set of states (excluding invalid transitions) in the SI protocol. For example, II-IS in PSI represents {III-IIS, III-ISI, III-ISS} in SI protocol. Therefore, if we traverse II-IS in PSI protocol, we can guarantee that we have covered one of the transitions in {III-IIS, III-ISI, III-ISS} in SI protocol. In other words, we mark {III-IIS, III-ISI, III-ISS} as similar transitions, and want to cover the representative of the three transitions within verification budget.



A SI protocol with 3 cores.

B PSI protocol with 3 cores and permutation group $G = (0,1)(2)$.

Figure 6-4. The original state space and its corresponding quotient space of SI with 3 cores.

We can prove similar arguments for PMSI vs MSI protocol. However, it is important to note that the state space of PMESI, PMOSI and PMOESI are no longer exactly the same as that of MESI, MOSI and MOESI protocols, respectively. This is because there are more transitions in the quotient protocol than the original one. For example, in a system with 4 nodes and permutation group $G = (0,1)(2,3)$, IIEI to IISS, IISO to IISI and IIIM to IISO will look like IE to IS, IO to IS, IM to IO, respectively, from the state space of quotient protocols. We call these transitions as *extra transitions*. Fortunately, the number of extra transitions is only $O(\alpha)$, which does not change the asymptotic size of the generated trace.

The total number of transitions can be computed by adding the extra transitions to the original transitions. The results are shown in Figure 6-5. As we can see, the number of transitions increase exponentially with the number of orbits α.

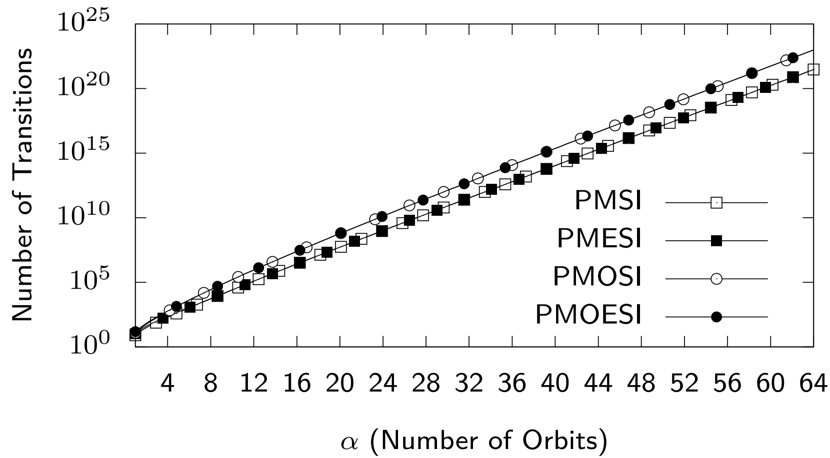


Figure 6-5. Complexity of quotient protocol with respect to number of orbits α.

The basic idea for verification using quotient protocol is to mark certain states as equivalent, i.e., acting permutation group G to partition the original nodes into α orbits, then define state on orbits. Transitions in quotient space are representatives of similar transitions in equivalence classes. When choosing α to be equal to the number of cores, full coverage is guaranteed. However, the number of total transitions grows so quickly that for large number of cores, it is unrealistic to verify all transitions, even using directed tests (one-to-one mapping between transitions and instruction sequences). Our quotient protocol identifies equivalence classes and selects the transitions to trade-off between transition coverage and validation time. For fixed number of cores, choosing larger number of orbits (α) means covering exponentially more representative transitions in the original protocol space, but it comes at the cost of increased validation effort. If we can cover all states and transitions in the quotient protocol with a test suite, the same test suite should be able to cover the most important transitions in the original protocol. The advantage of using orbits lies in the flexibility of grouping "similar" states. The way of forming orbits can be changed based on the verification budget and the

functionality of the cores. In order to increase the probability of covering the transitions of an important node, we may construct one orbit containing the important core, and group the rest randomly.

To illustrate how to perform test generation using quotient protocol, we start our discussion from SI protocol. For simplicity, we assume that the n nodes are evenly partitioned into α orbits, i.e., choose $G = (0, ..., k-1)(k, ..., 2k-1)...((\alpha-1)k, ..., n-1)$ where $k = \lceil n/\alpha \rceil$. We use $R(\boldsymbol{s})$ to represent the global orbit state of $\boldsymbol{s}$, where p^{th} element of $R(\boldsymbol{s})$ is

$$R_p(\boldsymbol{s}) = \begin{cases} S & \exists\, pk \leq i < (p+1)k,\ s_i = S \\ I & otherwise \end{cases}$$

It is easy to see that $R(\boldsymbol{s})$ has α elements and the pth element of $R(\boldsymbol{s})$ is shared if and only if there exists i such that $pk \leq i < (p+1)k$ and s_i is shared. Conceptually, quotient protocol PSI reduces the number of states by performing an "or" operation per k nodes in the original global state. The transitions in PSI are "abstract" transitions in the quotient state space. They cannot be directly executed or simulated in the actual design. Therefore, we design Algorithm 4 to generate corresponding feasible transitions for original protocol SI.

Algorithm 4 Test generation for quotient SI protocol with n cores

```
 1: procedure CreateTestsSI(n)
 2:     for r = 0 to α − 1 do
 3:         VisitHypercube(α, r)
 4:     end for
 5: end procedure
 6: procedure VisitHypercube(m, r)
 7:     p = (m + r) mod α
 8:     q = rand(k)
 9:     Output "load(pk + q)"
10:     for i = 1 to m − 1 do
11:         VisitHypercube(i, r)
12:     end for
13:     Output "evict(pk + q)"
14: end procedure
```

Algorithm 4 introduces randomness in "load" and "evict" operations using $rand(k)$ which returns i.i.d. random integers uniformly distributed between 0 and $k-1$. It is introduced to provide fairness among equivalent states. If we view the generated transition sequence from the state space of PSI, the sequence corresponds to a deterministic Euler tour of PSI's state space (Theorem 6.2). The randomness introduced by $rand(k)$ does not affect the transition, because $pk + rand(k)$ actually belongs to the same orbit regardless of the return value of $rand(k)$. Therefore, the generated sequence covers the entire state space of PSI with no wasted transitions.

Theorem 6.2. *The test sequence constructed by Algorithm 4 does perform an Euler tour of quotient protocol PSI's state space.*

The space complexity of Algorithm 4 is linear with the number of orbits α, because this algorithm requires a stack with at most $\alpha - 1$ levels. The time complexity is linear to the number of transitions $\alpha * 2^{\alpha}$. Clearly, it is asymptotically faster than on-the-fly approach [35] which has time complexity of $O(n2^n)$. This is obvious considering that PSI has asymptotically smaller state space with only 2^{α} states.

Similarly, randomization within orbits can be applied to generate efficient transition sequence with minimum wasted transitions for PMSI. Although PMESI, PMOSI and PMOESI are no longer strict MESI, MOSI and MOESI protocols, respectively, our test generation algorithms for MESI, MOSI and MOESI protocols can also be modified to support the quotient version by taking care of the extra transitions with additional efforts. As the number of extra transitions is only $O(\alpha)$, the asymptotic size of the generated traces does not change.

6.4 Experiments

6.4.1 Experimental Setup

To analyze the effectiveness of our proposed test generation framework, we conducted a number of experiments using Gem5 simulator [114] with Ruby memory subsystem. As we generate our test vectors to cover cache states and transitions using a certain path, correctness is verified by checking that the simulation using these vectors traverses the cache states and

Table 6-1. Gem5 simulation parameters

parameter	value
architecture	X86
cpu type	timing
clock frequency	1GHz
ruby	true
instruction cache size (L1)	4kB
data cache size (L1)	4kB
L1 hit latency	2ns
cache line size	4
memory size	4GB
number of cores	8, 16, 32, 64
debug flag	ProtocolTrace

transitions following this exact same path. Ruby memory subsystem implements MESI and MOESI cache coherence protocols by default, and provides interface to define other protocols. The detailed parameters for the simulation is shown in Table 6-1.

The overview of our evaluation framework is shown in Figure 6-6. We first use our test generation algorithms to generate the load/store/evict sequence and expected state sequence. The output sequence is fed into a program (TestSeqRunner in Figure 6-6) that we have designed to run inside the Gem5 and Ruby framework. TestSeqRunner is compiled using gcc with m5threads (to support barriers to synchronize all the threads) to run in Gem5. For N-core system, we create $N + 1$ threads: the main thread to read from the output of our algorithm, and the other N threads (one in each core) to execute the designated load/store instructions. For example, when the main thread gets an instruction "0, load" from the test sequence, it will set $current$ to be the pid of thread 1 (run in core 0). When the first *pthread_barrier_wait* of main thread is executed, it will wait on the second *pthread_barrier_wait*. At the same time, all the other threads will execute the *if* statement after their first *pthread_barrier_wait*. Only thread 1 will execute the load instruction as $current$ matches its pid. After all the threads finish the *if* statement, the main thread will move on and read the next instruction.

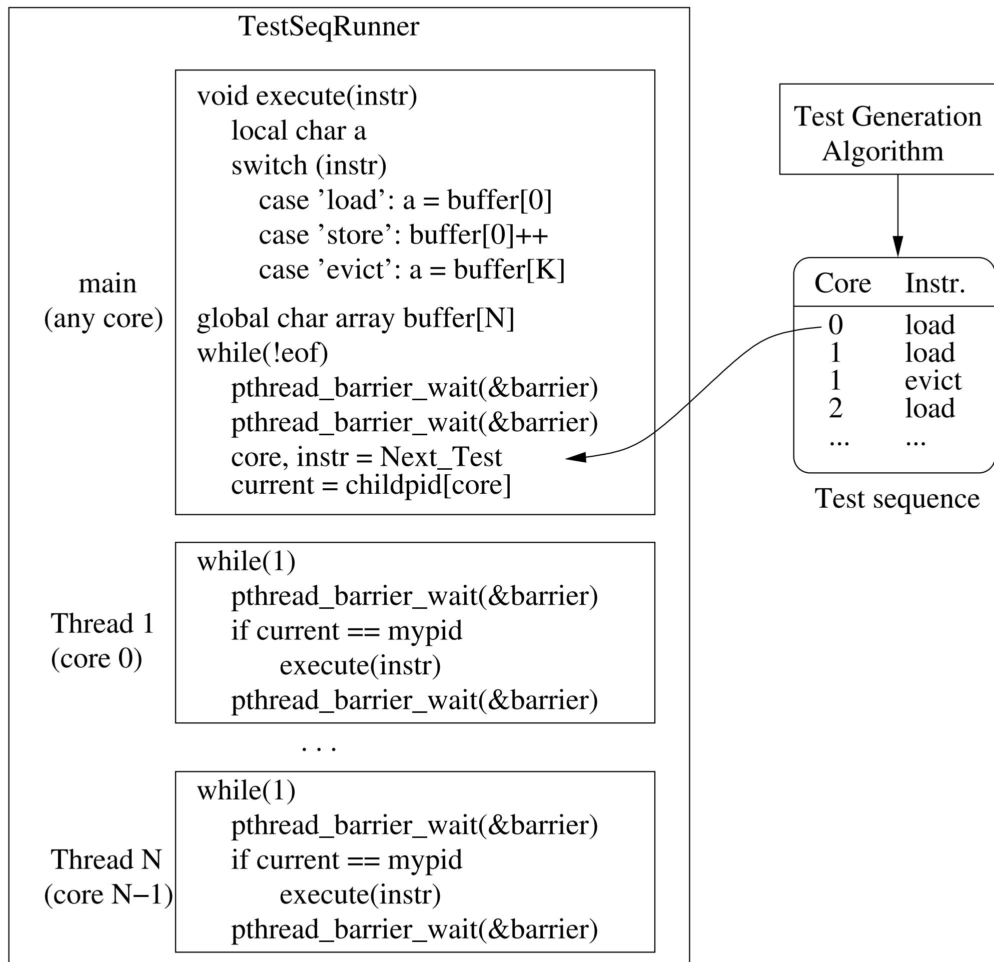


Figure 6-6. Evaluation framework of our experiment.

We monitor the cache behaviors with respect to a certain memory location. We first initialize an array that is larger than our cache. As shown in Figure 6-6, we first define a global char array $buffer$, and $buffer[0]$ is our location of interest. The *load* and *store* are done by reading from and writing to $buffer[0]$, respectively. While the *evict* operation is achieved by loading a different memory address $buffer[4096]$ which is also mapped to the same location in the cache as the cache block under test. Since the cache size is 4K bytes, having the least significant 12 bits being the same ensures that $buffer[0]$ and $buffer[4096]$ map to the same cache block. The protocol trace of Gem5 contains intermediate states, which are not considered in our approach. So we remove these states before comparing with the expected outputs.

The remainder of this section is organized as follows. First, we present coverage results (by choosing the number of orbits (α) to be the number of cores) using on-the-fly test generation techniques outlined in Section IV. Then, we present the results using quotient space (by varying α) to trade-off between functional coverage and verification efforts.

6.4.2 Test Generation for Quotient Protocol

Transition coverage in quotient state space is an effective way for test size reduction. With our quotient space based test generation techniques, verification engineers can pick the number of orbits α according to their verification budget and protocol complexity without losing any important transitions.

To compare the transition coverage in the original state space with different number of orbits (α), we vary $\alpha = 4, 8, 12, 16$ for MESI protocol with 32 cores. For $\alpha = 4, 8, 16$, we simply divided all the cores evenly into α orbits. While for $\alpha = 12$, we choose 4 orbits with 4 cores, and 8 orbits with 2 cores. The test generation time and coverage in the original space are shown in Figure 6-7. Note that the coverage and test generation time grow exponentially with the number of orbits. As shown in Example 5, our approach guarantees the selection of important transitions and omits similar transitions by utilizing equivalence classes.

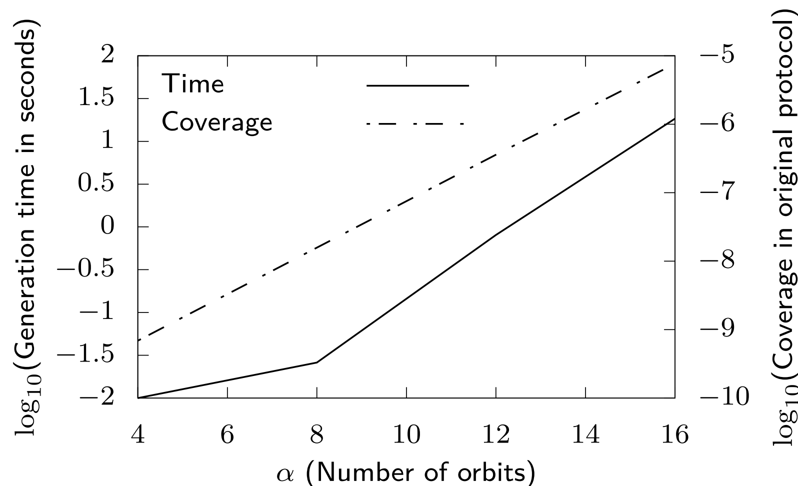


Figure 6-7. Test generation time (left y-axis) and coverage (right y-axis) in the original space (MESI with 32 cores) of PMESI protocol with different number of orbits.

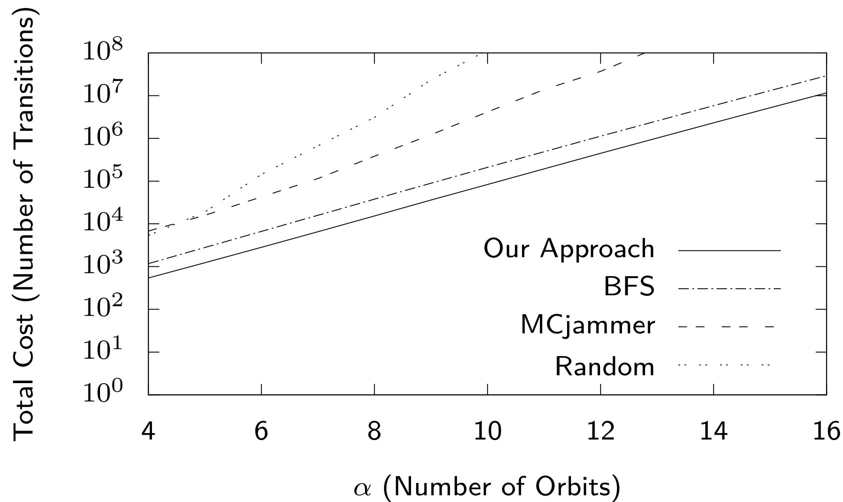


Figure 6-8. Total cost vs. number of orbits (α) for PMESI protocol with 64 cores.

To select the suitable α for a given verification budget, we first gather the total cost to achieve full coverage in the quotient space with different number of orbits (α). An important feature of our quotient space protocol is that it can be applied on top of any existing test generation algorithms. We configure BFS, MCjammer and Random algorithm to run on the quotient protocols. For the MCjammer and Random algorithm, the mean of multiple measurements are used to reduce the variation introduced by randomization. The experimental result is shown in Figure 6-8. As expected, choosing a lower α would require less transitions to achieve 100% coverage in quotient space, but achieves exponentially smaller coverage in the original protocol space as shown in Figure 6-7. For the same α, our method requires the least amount of cost to achieve full coverage in quotient space, and outperforms other approaches by several orders-of-magnitude. For example, for $\alpha = 8$, our approach requires about 10^4 transitions, while BFS requires twice as much, and MCjammer and Random algorithm require about 10^5 and 10^6 transitions, respectively.

In the experiment, we consider MESI with 64 cores. We did not provide results for other coherence protocols since they lead to similar observations in terms of reduction in validation effort. Let us assume that the verification budget is 10^7, i.e., total number of transitions cannot exceed 10^7. Based on Figure 6-8, our quotient protocol PMESI chooses $\alpha = 15$.

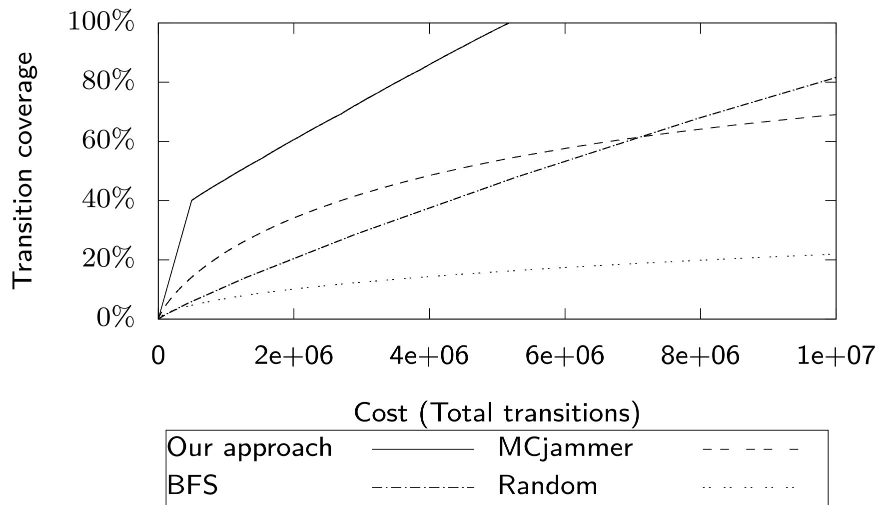


Figure 6-9. Transition coverage vs. time cost for PMESI protocol with 64 cores and 15 orbits.

Now, we would like to compare transition coverage of our test generation approach with other approaches on quotient protocols given the same α. Figure 6-9 shows the relation between transition coverage and testing cost on the quotient protocol. As we can see, our test generation approach achieves full coverage quickly taking advantage of Euler traversals, while none of the existing approaches can achieve full coverage within 10^7 transitions budget and 15 orbits. Clearly, our test generation approach on quotient protocol significantly outperforms the existing test generation approaches by providing higher design quality (coverage) within specific verification budget.

6.5 Summary

In this chapter, we presented quotient space based scalable test generation algorithms that can trade-off between functional coverage and verification effort. Quotient space guarantees selection of important transitions by utilizing equivalence classes, and omits only similar transitions to provide scalable test generation framework. Our experimental results demonstrated the effectiveness of our approach on systems with many cores and complex cache coherence protocols, making it suitable for future multicore architectures.

CHAPTER 7
SCALABLE ACTIVATION OF RARE TRIGGERS

Hardware Trojans are malicious modifications incorporated in simple Integrated Circuits (ICs) or complex System-on-Chip (SoC) designs [24] from the outsourcing and integration of third-party hardware IPs. These small malicious modifications hide behind complex SoCs with millions of gates. Hardware Trojans are designed in such a way that they are inactive under majority of normal usage conditions and be activated in extremely rare conditions. However, once triggered, they are able to alter the original functionality or leak secret information to the outside. Therefore, detection of hardware Trojans from untrusted manufacturers is an important step in security validation.

A Trojan consists of a *trigger* and a *payload*. In order to evade conventional testing such as simulation-based validation using random or constrained-random tests, an intelligent adversary is likely to design hardware Trojans in a way that they can only be triggered by extremely rare circuit conditions [12] called *trigger conditions*. Trigger conditions can be combinational when the Trojan could be triggered using a combination of data buses satisfying their rare values, or sequential when the Trojan could be triggered by a number of times a rare value is satisfied. The *payload* decides the effect of the Trojan after being triggered. By altering the value of payload, undesired functional behavior can happen, such as changing the content of memory locations, or leaking the secret information to the outside. A simple 3-triggered Trojan is shown in Figure 7-1, where signals A, B, C and D are called rare nodes with their rare values being 0, 1, 0 and 1, respectively. Assume each value is satisfied for less than 10% of total time, the trigger condition is activated with a probability less than 10^{-3} if these three signals are relatively independent. Conventional testing using millions of random tests is expected to fail in activating trigger condition with more rare-to-activate signals.

To address the fundamental challenge of activating rare triggers, we propose a new test generation paradigm for Trigger Activation by Repeated Maximal Clique sampling (TARMAC). We solve the trigger activation problem by mapping it to the problem of covering maximal

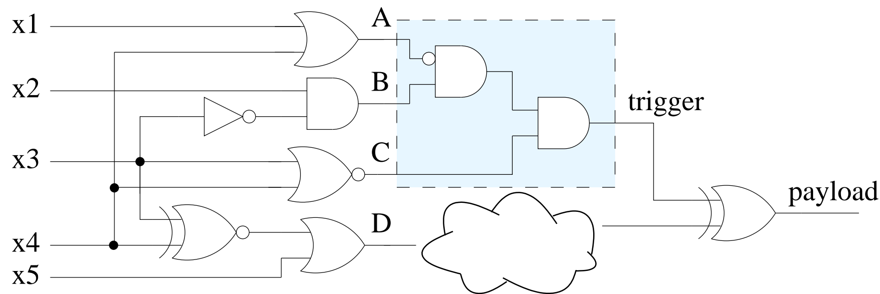


Figure 7-1. A simple combinational Trojan with 3 triggers.

cliques in a graph. Then, we utilize a satisfiability modulo theories (SMT) solver to construct a test corresponding to each maximal clique activate extremely rare trigger conditions that can be covert during traditional validation. The major contributions of this chapter are as follows:

1. To the best of our knowledge, our approach is the first attempt to map trigger activation problem to maximal clique cover problem. We prove that the test vectors generated by covering maximal cliques are complete and compact considering trigger coverage and test length.
2. We propose an efficient test generation algorithm for Trigger Activation by Repeated MAximal Clique sampling (TARMAC).
3. We outline an algorithm to support concurrent execution of time-consuming computations to improve the scalability of TARMAC.
4. Experimental results demonstrate that TARMAC outperforms the state-of-the-art test generation techniques by several orders-of-magnitude for extremely rare-to-activate trigger conditions in large designs.

The rest of this chapter is organized as follows. In Section 7.1, we motivate the need for this work by highlighting the drawbacks of $N-$detect paradigm as well as the limitations of the state-of-the-art test generation approaches. Section 7.2 describes our proposed test generation framework. Section 7.3 presents the experimental results. Finally, Section 7.4 concludes this chapter.

7.1 Motivation

$N-$detect paradigm has been successful in both logic testing [12, 55] and side-channel analysis [6]. $N-$detect paradigm requires the test set to activate each rare signal N times and is statistically effective for trigger activation given "sufficiently" large N [12]. The

probability of activating trigger conditions will significantly decrease when the trigger condition is composed of very rare signals. It is expected that increasing N can increase the chances of hitting trigger conditions. However, larger N will significantly deteriorate the test generation performance and increase the required test length. MERO incorporated $N-$detect idea [12] with deterministic flipping method as shown in Algorithm 5, and the quality of generated test vectors is highly dependent on the quality of the initial random vectors. MERO has the following two major problems that make it ineffective for activating hard-to-detect trigger conditions in large designs.

Algorithm 5 MERO [12]

```
1:  procedure MERO(R, N)
2:      Tests = {}
3:      simulate design with R random vectors
4:      sort random vectors by the number of rare signal hits
5:      for each vector u in random vectors do
6:          for each bit u_i in u do
7:              Flip u_i and simulate the design
8:              if N−detect criteria does not improve then
9:                  reverse flipping
10:             end if
11:         end for
12:         Tests = Tests ∪ u, if u improves N-detect criteria
13:     end for
14: end procedure
```

Scalability Problem: Although MERO claimed to implement $N-$detect, the generated test vectors cannot guarantee that each rare signal is activated at least N times. With the same configuration ($R = 100K, N = 1000$) for the same ISCAS benchmarks [12] and ($R = 1M, N = 1000$) for MIPS processor from [96], we examined the number of times each rare signal is activated by MERO as shown in Figure 7-2. There are some extremely rare signals (outliers below the green line) that are almost never activated in most benchmarks, while some signals (outliers above the green line) are activated more than N times. To ensure $N-$detect for all rare signals, the number of initial random vectors should be extremely large even for small benchmarks. To show how the number of random vectors affects $N-$detect

in MERO, we set $N = 1000$ and vary the number of random vectors. The percentage of rare signals that are activated more than 1000 times is shown in Figure 7-3. As expected, the percentage of $N-$detect rare signals grows rapidly when the number of random vectors is small, but very slowly beyond a specific number. It is expected that for large designs, billions of random vectors are required to satisfy $N = 1000$. MERO requires *one simulation per bit flipping*, the total number of simulations would be in the order of billions or trillions, which makes this approach impractical for large designs.

Poor Trigger Coverage: MERO uses a vague notion of N being "sufficiently" large to ensure high trigger coverage. In fact, MERO simply selected $N = 1000$ in [12] for all benchmarks. Despite the fact that all rare signals are activated at least 1000 times in the small benchmark, such as c5315, (see Figure 7-2), the trigger coverage is only 50.6% (see Section 7.3.3). In other words, $N = 1000$ is not "sufficiently" large for such a small benchmark. For larger designs with more trigger points and lower rareness threshold, larger N is required to reach even a reasonable coverage by MERO, which needs drastically larger number of initial random vectors as discussed above, making scalability issue even worse.

Given the poor trigger coverage and scalability issue of MERO and $N-$detect, new paradigms are need to solve trigger activation problem. In this chapter, we address the fundamental challenge of trigger activation by mapping it to clique cover problem and finding the test patterns to cover maximal cliques, as outlined in the next section.

7.1.1 Maximal Clique Problem

Clique decision problem is listed as one of Karp's 21 NP-complete problems [115]. Maximal clique problem [116] is the problem that given a set of vertices and their connectivity, find the maximal clique that no other vertex can be added. As proved by Moon and Moser [117], the number of maximal cliques is $O(3^{n/3})$ for n vertices in the worst case. Therefore, the effort of listing all maximal cliques is exponential to the number of vertices. Many efficient and parallel approaches [118–120] exist in practice. Bron–Kerbosch algorithm [118] is a widely used approach to list all maximal cliques in a graph. It is a recursive procedure that

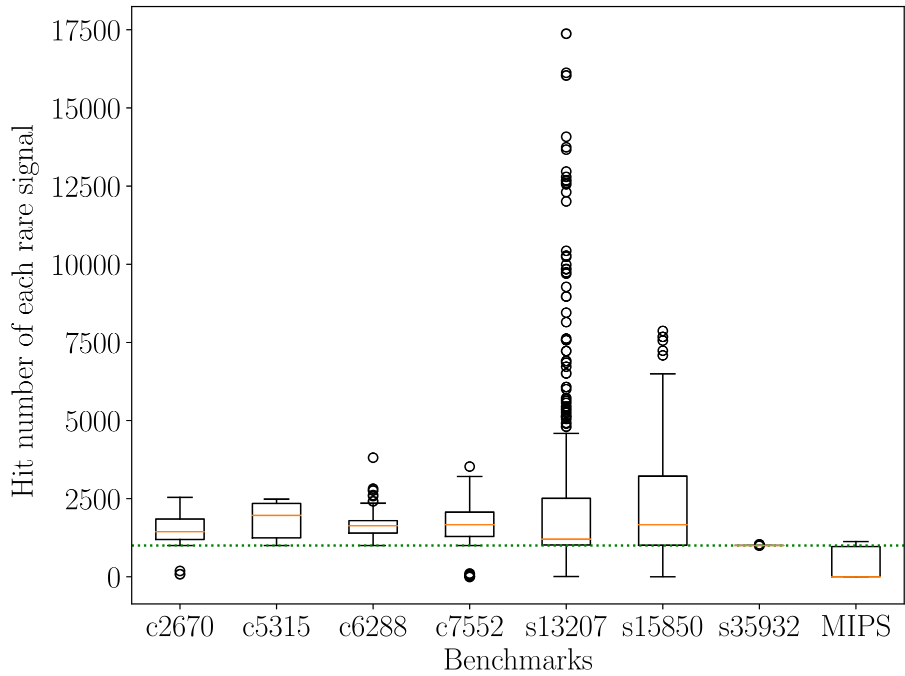


Figure 7-2. The number of times each rare signal is activated by by MERO.

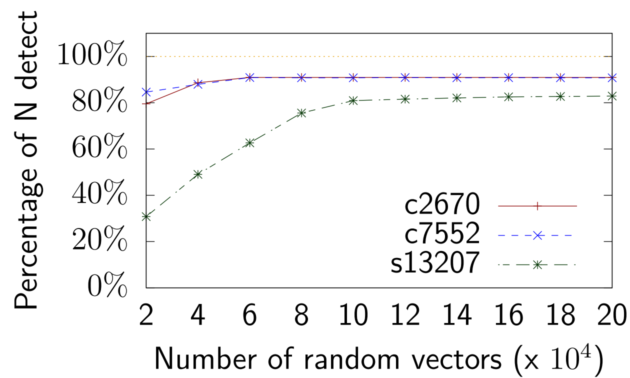


Figure 7-3. The percentage of rare signals that are activated at least N times by MERO.

keeps track of three disjoint sets R, P and X, representing constructed clique, candidate vertices and excluded vertices, respectively. The existence of X ensures that maximal cliques are not repeated. Each recursive call adds one vertex from P to R and reports maximal clique when P and X are both empty. The worst-case running time matches the largest number

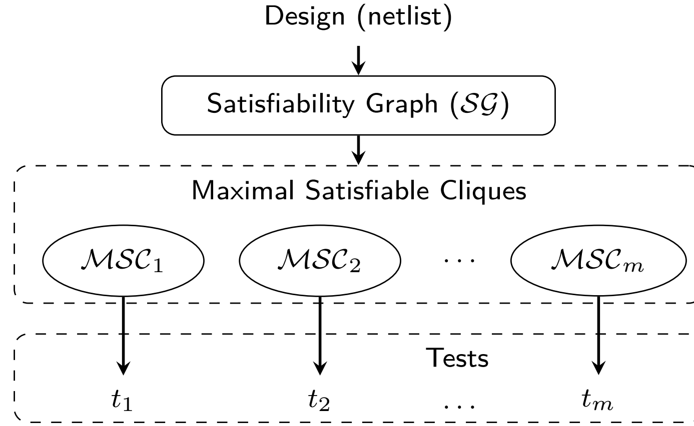


Figure 7-4. Overview of our proposed (TARMAC) paradigm.

of maximal cliques in [117]. In this chapter, we utilize maximal clique to solve the trigger activation problem as described in Section 7.2.

7.2 Scalable Activation of Rare Triggers

In this section, we propose a new test generation paradigm (TARMAC) to solve trigger activation problem by mapping it to maximal clique cover problem, as shown in Figure 7-4. Our approach first constructs a satisfiability graph based on the design (e.g., gate-level netlist). Next, it finds maximal satisfiable cliques ($\mathcal{MSC}$s) in the satisfiability graph. Finally, it utilizes a SAT solver [121] to generate one test for each maximal satisfiable clique. This section is organized as follows. We first define a few terms that are used in the chapter. Next, we describe the mapping of trigger activation to clique cover problem and prove that the generated test set is complete and compact. Finally, we describe three test generation algorithms to find and cover maximal satisfiable cliques using directed clique enumeration (Algorithm 2), random sampling and lazy construction of satisfiability graph (Algorithm 3), and scalable TARMAC with multi-threaded execution (Algorithm 4), respectively.

7.2.1 Definition and Notations

Without any loss of generality, in this chapter, we consider gate-level implementation of designs. We call the graph level representation of the design *a Design Graph* ($\mathcal{DG}$), where each vertex represents a signal and each edge represents the connectivity (via a gate). For each

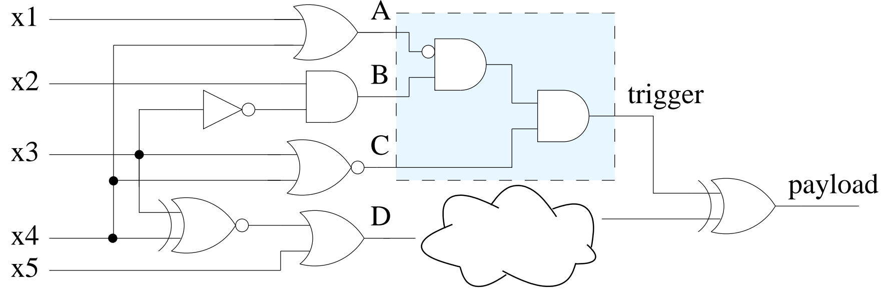


Figure 7-5. A hardware Trojan with a trigger condition constructed by three rare signals.

signal, we compute its logic expression (le) from its corresponding logic cone. For example, the logical expression of vertex A in Figure 7-5 is $A.le = x_1 \vee x_4$. For sequential circuits, we assume that design-for-debug architecture (e.g., scan chain) exists and the logic expression can be formulated using any register values.

For each design, trigger conditions can be constructed from a subset of its signals and their corresponding rare value rv, which we refer as ***potential trigger signals (*PTS*)***. PTS could be any subset of signals. In [12], PTS is the set of rare signals that are used to construct hard-to-activate trigger condition. A trigger signal is *activated* if it satisfies its rare value. We define *satisfiability graph* as follows.

Definition 7.1. *A* **Satisfiability Graph** *($\mathcal{SG}$) consists of vertices representing PTS and their satisfiability connections, $\mathcal{SG} = \{\mathcal{V}, \mathcal{E}\}$ where $\mathcal{V} == PTS$. If $(u.le == u.rv) \wedge (v.le == v.rv)$ is satisfiable, then there exists an edge between u and v, i.e., $u \in \mathcal{E}(v)$ and $v \in \mathcal{E}(u)$.*

Let us consider the example in Figure 7-5 with four PTS (A, B, C, D) and their corresponding rare values (0, 1, 1, 0). To construct the satisfiability graph for this example, we need to use their logical expressions described above and determine their connectivity. To find out if there is an edge between any two vertices, we check if any input (test) pattern exists that satisfies both rare values. For example, the edge between A and B exists since input pattern 01000 satisfies the condition $(x_1 \vee x_4 == 0) \wedge (x_2 \wedge \neg x_3 == 1)$. In other words, 01000 can activate both A and B at the same time with their respective rare values. On the other hand, there is no input pattern that satisfies $(\neg(x_3 \vee x_4) == 1) \wedge (\neg(x_3 \oplus x_4) \vee x_5 == 0)$, i.e., there is no edge between C and D. The constructed satisfiability graph is shown in Figure 7-6

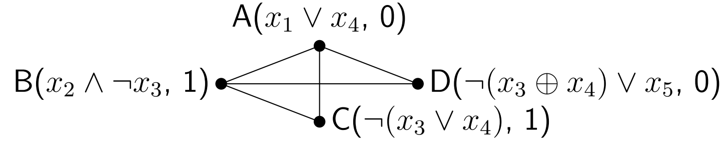


Figure 7-6. The satisfiability graph with 4 PTS (A,B,C,D) from Figure 7-5, with logic expressions and rare values in parentheses.

(logic expressions and rare values are shown inside parentheses). It is easy to see that $\mathcal{SG}$ is an undirected graph.

7.2.2 Mapping Trigger Activation to Clique Cover Problem

A fundamental contribution of this chapter is to show that trigger activation problem can be mapped to clique cover problem. First, we show that any valid trigger condition forms a clique in satisfiability graph $\mathcal{SG}$.

Lemma 1. *For any valid trigger condition with k rare signals $\{v_1, v_2, ..., v_k\}$, the vertices $\{v_1, v_2, ..., v_k\}$ form a $k-$clique in the satisfiability graph $\mathcal{SG}$.*

Proof. We prove Lemma 1 by contradiction. Assume that there is no edge between v_i and v_j. By definition, condition $(v_i.le == v_i.rv) \wedge (v_j.le == v_j.rv)$ is not satisfiable. Therefore, there will be no test that can activate v_i and v_j together, invalidating the trigger condition. Since there is an edge between any pair of vertices, $\{v_1, v_2, ..., v_k\}$ form a $k-$clique in the satisfiability graph $\mathcal{SG}$. □

Note that it is possible to have a clique in the satisfiability graph that does not represent a valid trigger condition. For example, consider the clique ABD in Figure 7-6. There is no input pattern that satisfies the condition $(x_1 \vee x_4 == 0) \wedge (x_2 \wedge \neg x_3 == 1) \wedge (\neg(x_3 \oplus x_4) \vee x_5 == 0)$, although there are edges between any two of the three vertices. In other words, ABD forms a clique in $\mathcal{SG}$, but it does not represent a valid trigger condition. Clearly, an adversary will not use it as a Trojan trigger since it cannot be triggered. For the ease of illustration, we define *satisfiable clique* in Definition 7.2. The relationship between satisfiable cliques and valid trigger conditions is shown in Lemma 2 and Lemma 3.

Definition 7.2. *A satisfiable clique $\mathcal{SC}$ is a clique in a satisfiability graph $\mathcal{SG}$, where all the vertices of $\mathcal{SC}$ can be activated by the same input vector.*

Lemma 2. *Any valid trigger condition can be represented as a satisfiable clique $\mathcal{SC}$ in satisfiability graph $\mathcal{SG}$.*

Proof. Lemma 1 proves that any valid trigger condition forms a clique in $\mathcal{SG}$. Validity of this trigger condition ensures that all vertices can be activated by the same input vector. By Definition 7.2, this clique is a satisfiable clique. □

Lemma 3. *Any satisfiable clique $\mathcal{SC}$ in satisfiability graph $\mathcal{SG}$ represents a valid trigger condition.*

Proof. For any satisfiable clique, all its vertices can be activated by a test vector by Definition 7.2. Thus, these vertices represent a valid trigger condition. □

Finally, we explore the mapping from the set of valid trigger conditions to the set of satisfiable cliques in Theorem 7.1. It points out a new way to solve trigger activation problem, i.e., finding test vectors to cover satisfiable cliques in a satisfiability graph.

Theorem 7.1. *The mapping between the set of valid trigger conditions and the set of satisfiable cliques is a bijection.*

Proof. As different trigger conditions consist of at least one different rare signal, the corresponding satisfiable cliques have at least one different vertex. Hence, no two valid trigger conditions map to the same satisfiable clique, i.e., the mapping from the set of valid trigger conditions to the set of satisfiable cliques is an injection from Lemma 2. Similarly, we can prove that the mapping from the set of satisfiable cliques to the set of valid trigger conditions is also an injection from Lemma 3. Therefore, we have a one-to-one mapping between these two sets. □

7.2.3 Directed Test Generation Scheme

Lemma 4. *If one test vector can satisfy a satisfiable clique, all its subgraphs can be satisfied by the same test vector.*

Proof. Let $\mathcal{R}$ be a subgraph of a satisfiable clique $\mathcal{SC}$. By Definition 7.2, all vertices in $\mathcal{SC}$ can be satisfied by the same test vector t. All vertices of $\mathcal{R}$ are inherently satisfiable by t since the vertices of $\mathcal{R}$ is a subset of the vertices of $\mathcal{SC}$. □

Lemma 5. *A subgraph of a satisfiable clique is also a satisfiable clique.*

Proof. For any satisfiable clique $\mathcal{SC}$, its subgraph $\mathcal{R}$ is a clique as $\mathcal{SC}$ is a clique. By Lemma 4, $\mathcal{R}$ is satisfiable. By definition, $\mathcal{R}$ is a satisfiable clique. □

Therefore, if we are able to find a test vector that can satisfy a clique, it is not necessary to generate any more test for all the trigger conditions represented by its subgraphs. Clearly, *the most profitable test vector is the one that can satisfy the largest clique*. Similar to cliques in graph theory, we define a *maximal satisfiable clique* in Definition 7.3.

Definition 7.3. *A maximal satisfiable clique ($\mathcal{MSC}$) is a satisfiable clique to which no more vertices can be added.*

Let $\{\mathcal{MSC}_1, \mathcal{MSC}_2, \ldots, \mathcal{MSC}_n\}$ represents the complete set of maximal satisfiable cliques, where n is the total number of maximal satisfiable cliques. For example, $\{\mathcal{MSC}_1 =$ ABC, $\mathcal{MSC}_2 =$ AD, $\mathcal{MSC}_3 =$ BD$\}$ represents the complete set of maximal satisfiable cliques in Figure 7-6. Next, we prove that the set of test vectors that activate all elements in $\{\mathcal{MSC}_i\}$ is optimal in activating all possible trigger conditions in the design.

Theorem 7.2. *Let t_i be an input pattern that activates the corresponding maximal satisfiable clique $\mathcal{MSC}_i$. Then, the test set $T = \{t_i\}$ is complete and compact, i.e., it is the shortest test set that can activate all valid trigger conditions.*

Proof. We first prove the completeness of our test set. For any valid trigger condition, it forms a satisfiable clique $\mathcal{SC}$ by Theorem 7.1. By definition of maximal satisfiable cliques, there exists some maximal satisfiable clique $\mathcal{MSC}_i$ such that $\mathcal{SC} \subset \mathcal{MSC}_i$. As $t_i \in T$ satisfies $\mathcal{MSC}_i$, it inherently satisfies satisfiable clique $\mathcal{SC}$ by Lemma 4. As T can satisfy all elements in $\{\mathcal{MSC}_i\}$, it can satisfy any valid trigger condition.

Now, we prove that the test set is compact. It is easy to see that any two maximal satisfiable cliques can never be activated by the same test vector, otherwise, they form a larger satisfiable clique which contradicts the definition of maximal satisfiable clique in Definition 7.3. As any maximal satisfiable clique represents a valid trigger condition by Lemma 3, a test set that can activate all these trigger conditions need at least $|\{\mathcal{MSC}_i\}|(=|T|)$ test vectors. Hence, no test set that satisfies all trigger conditions can be shorter than T. □

As a result, the problem of test generation for trigger activation can be reduced and mapped to the problem of finding maximal satisfiable cliques and generate directed test for them. Based on Theorem 7.2, the generated test vectors are the optimal solution considering both trigger coverage and test length. For the example in Figure 7-6, we need exactly three tests - t_1 (01000), t_2 (01100) and t_3 (11010) to activate maximal satisfiability cliques ABC, AD, and BD, respectively.

7.2.4 Test Generation Algorithms

In this section, we present two test generation algorithms to generate test patterns by covering maximal satisfiability cliques. Algorithm 6 (Section 7.2.4.1) is guaranteed to generate the complete test set (covers all the trigger conditions) but is not scalable since it requires enumeration of potentially exponential number of $\mathcal{MSC}s$. In addition, it has the bottleneck of construction of the full satisfiability graph. This algorithm is suitable when only a small number of rare signals are in a design. To address the scalability issue, Algorithm 7 (Section 7.2.4.2) replaces the enumeration problem by randomly sampling $\mathcal{MSC}s$, and it performs lazy construction of the satisfiability graph. It is significantly faster and effective, but cannot guarantee completeness. The reminder of this section describes these algorithms.

7.2.4.1 Test Generation using Clique Enumeration

Based on Theorem 7.2, we propose our first straightforward test generation algorithm based on *clique enumeration*. The main steps of this approach are shown in Algorithm 6. The procedure of $TestGeneration$ first parses and constructs the design graph ($\mathcal{DG}$) from the gate-level netlist, and computes all the logic expressions. Then, the vertices of satisifiability

graph ($\mathcal{SG}$) are initialized from PTS and the edges are constructed after testing satisfiability of any two vertices ($ConstructSatisfiabilityGraph$). Next, Bron-Kerbosch algorithm [118] is applied to find all maximal cliques in $\mathcal{SG}$. For every clique $\mathcal{C}$ found in line 6, we need to find all maximal satisfiable cliques inside $\mathcal{C}$. Finally, test vectors are generated for each maximal satisfiable clique.

Algorithm 6 Test Generation by Clique Enumeration

```
 1: procedure TestGeneration(circuit netlist CN, potential trigger signals PTS)
 2:     DG = ConstructDesignGraph (CN)
 3:     Compute logic expressions for PTS in DG
 4:     SG = ConstructSatisfiabilityGraph(DG, PTS)
 5:     Clique set CS = Bron-Kerbosch(SG)
 6:     for each clique C in CS do
 7:         for each maximal satisfiable clique in C do
 8:             Use SMT solver to generate a test vector t_i for it
 9:         end for
10:     end for
11:     return Tests = {t_1, t_2, ..., t_n}
12: end procedure

13: procedure ConstructSatisfiabilityGraph(DG, PTS)
14:     SG.V = PTS, SG.E(u) = {}
15:     for u, v ∈ SG.V do
16:         SAT expression S = (u.le == u.rv) ∧ (v.le == v.rv)
17:         if satisfiabile(S) then
18:             SG.E(v) = SG.E(v) ∪ {u}
19:             SG.E(u) = SG.E(u) ∪ {v}
20:         end if
21:     end for
22:     return SG
23: end procedure
```

Next, we prove that the generated test vectors are complete. For any maximal satisfiable clique, it must be a subgraph of some maximal clique $\mathcal{C}$ enumerated by Bron-Kerbisch [118]. Line 7 ensures that all maximal satisfiable cliques are found when we visit $\mathcal{C}$. By Theorem 7.2, the generated test vectors are complete.

This approach is effective in small designs, but it lacks the scalability due to the following three major bottlenecks:

1. The computational problem of finding all maximal cliques is NP-hard [122]. Although Bron–Kerbosch algorithm [118] is practical in finding all maximal cliques, it suffers from deep recursive function calls for large graphs with the worst running time $O(3^{n/3})$ [123], where n is the number of vertices.

2. Finding all maximal satisfiable cliques inside a large clique (e.g., more than 20 vertices) is difficult. A brute-force approach need to check the satisfiability of all possible combinations. The running time is exponential to the size of the clique.

3. Algorithm 6 also has the bottleneck of constructing the full satisfiability graph. When the number of vertices $|\mathcal{SG}.\mathcal{V}|$ is extremely large, checking if an edge exists between two vertices requires approximately $|\mathcal{SG}.\mathcal{V}|^2/2$ calls of the SMT solver, which can be prohibitive in terms of debug time.

7.2.4.2 Efficient Test Generation using Clique Sampling and Lazy Construction

To address both clique enumeration and satisfiability graph construction issues in Algorithm 6, we propose an on-the-fly technique (TARMAC) in Algorithm 7 that utilizes lazy construction of the satisfiability graph and random sampling of maximal satisfiable cliques. The random sampling makes TARMAC scalable to large designs with the cost of completeness. For each sampled maximal satisfiable clique, TARMAC generates one test vector for it.

Clique sampling is done by maintaining two sets of variables: cns to keep track of constraints that are satisfiable (represents vertices that are already found in a satisfiable clique), and P to represent candidate vertices that may potentially be added to the clique. Initially, cns is true and P contains all the vertices. We first randomly select and remove a vertex v from candidate set P. If cns can be augmented by $v.le == v.rv$, we put it into cns and remove all vertices in P that are not connected to v (line 16). It is easy to verify that cns represents a maximal satisfiable clique when P is empty. Parameter *VN* is used to control how many times we should sample maximal satisfiable cliques, i.e., the number of generated test vectors.

The complexity of full satisfiability graph construction is eliminated by lazy construction. As shown in Algorithm 7, initially every vertex is connected to every other vertices in line 3. Whenever we find two vertices unsatisfiable (line 17), we remove the edge between these two vertices. Lazy construction benefits large designs by generating test vectors as soon as

possible, with the cost of wasted SMT solver calls. If we look at the example in Figure 7-6, Algorithm 6 disconnects C and D before searching for cliques, while Algorithm 7 constructs a fully connected graph initially, which may introduce multiple wasted SMT solver calls in the clique sampling process involving C and D. These two vertices will be disconnected in line 17-19 only when they are selected as the first two vertices from P in line 13, with the probability of approximately $2/|\mathcal{SG}.\mathcal{V}|^2$. Statistically, the full graph will be constructed after $|\mathcal{SG}.\mathcal{V}|^2/2$ sampling.

Algorithm 7 Test Generation using Random Sampling and Lazy Construction (TARMAC)

1: **procedure** $TARMAC$(circuit netlist CN, potential trigger signals PTS, maxVectorNumber VN)
2: $\mathcal{DG}$ = ConstructDesignGraph (CN)
3: Compute logic expressions for PTS in $\mathcal{DG}$
4: $\mathcal{SG}.\mathcal{V} = PTS$, $\mathcal{SG}.\mathcal{E}(u) = \mathcal{SG}.\mathcal{V} \setminus \{u\}$
5: **for** $i = 1$ to VN **do**
6: t_i = CliqueSampling($\mathcal{SG}$)
7: **end for**
8: *return* $Tests = \{t_1, t_2, \ldots, t_{VN}\}$
9: **end procedure**

10: **procedure** $CliqueSampling$($\mathcal{SG}$)
11: constraints $cns = true$, $P = \mathcal{SG}.\mathcal{V}$
12: **while** P is not empty **do**
13: randomly pick and remove a vertex v from P
14: **if** satisfiabile($cns \wedge (v.le == v.rv)$) **then**
15: $cns = cns \wedge (v.le == v.rv)$
16: $P = P \cap \mathcal{SG}.\mathcal{E}(v)$
17: **else if** cns has one constraint $u.le == u.rv$ **then**
18: $\mathcal{SG}.\mathcal{E}(v) = \mathcal{SG}.\mathcal{E}(v) \setminus \{u\}$
19: $\mathcal{SG}.\mathcal{E}(u) = \mathcal{SG}.\mathcal{E}(u) \setminus \{v\}$
20: **end if**
21: **end while**
22: Use SMT solver to solve cns and *return* the test
23: **end procedure**

7.2.5 Scalable TRAMAC by Parallelization of Clique Sampling

By inspecting the process of clique sampling, we can see that this process is highly parallelizable. To further increase the efficiency of Algorithm 7, we add parallelism to clique

sampling, i.e., $TARMAC_p$, as shown in Algorithm 8. Instead of generating all VN test vectors in one thread, $TARMAC_p$ evenly splits the task into NT threads, where each thread generates a batch of $VN_p = VN/NT$ test vectors. In order to minimize the overlapped efforts of covering the same cliques by different threads, we feed a different random seed to each batch sampling (line 7 and 8). Then, each thread runs $batchSampling$ independently. It sets the random seed, and calling the modified version of clique sampling to generate VN_p test vectors. After a thread completes its job, the generated test vectors are appended to the list of final tests. Comparing the clique sampling in Algorithm 8 and Algorithm 7, the only differences are line 28 and 31, where mutex is used to safely update the edges of shared satisfiability graph $\mathcal{SG}.\mathcal{E}$. Except for this block, the data structures are either copied, e.g. $\mathcal{SG}.\mathcal{V}$ in line 21, or are only for reading, e.g., $\mathcal{SG}.\mathcal{E}$ in line 26. For efficiency consideration, a simple mutex is used to prevent multiple writing to $\mathcal{SG}.\mathcal{E}$, instead of a readers-writer lock. In other words, this simple mechanism allows multiple threads to read $\mathcal{SG}.\mathcal{E}$ (line 26) while one thread is writing to it. The only difference compared to a readers-writer lock is that simple mutex mechanism will read old version of $\mathcal{SG}.\mathcal{E}$ in line 26, which makes P to contain one redundant vertex. It is not critical since the redundant vertex will be removed in a future iteration anyway. When multi-core infrastructures are provided, $TARMAC_p$ can achieve high efficiency improvement over $TARMAC$ due to the parallelism of constraints solving.

7.2.6 Effectiveness of Random Clique Sampling

In Section 7.2.4, we introduced two algorithms, i.e., clique enumeration (Algorithm 6), and random clique sampling with lazy construction (TARMAC, Algorithm 7). As expected, random sampling cannot guarantee to find all maximal satisfiable cliques as clique enumeration. In this section, we show why random sampling is still effective.

Let us consider two scenarios shown in Figure 7-7, where the circles of C_1, C_2, C_3 represent maximal SAT cliques, and the octagon represents the 8-trigger condition. The only difference between Figure 7-7(a) and Figure 7-7(b) is the size of maximal SAT cliques. In the large clique scenario (Figure 7-7(a)), the average size of maximal SAT cliques is 200,

Algorithm 8 Parallelization of TARMAC

```
1:  procedure TARMAC_p(circuit netlist CN, potential trigger signals PTS,
    maxVectorNumber VN, number of threads NT)
2:      DG = ConstructDesignGraph (VN)
3:      Compute logic expressions for PTS in DG
4:      SG.V = PTS, SG.E(u) = SG.V \ {u}
5:      The number of vectors per thread VN_p = VN/NT
6:      for td = 1 to NT do    // NT threads
7:          seed = random()
8:          Create a new thread to execute batchSampling (SG, VN_p, seed)
9:          Append the generated tests to Tests
10:     end for
11:     return Tests = {t_1, t_2, ..., t_VN}
12: end procedure

13: procedure batchSampling(SG, VN_p, seed)
14:     random.seed(seed)
15:     for i = 1 to VN_p do
16:         t_i = CliqueSampling(SG)
17:     end for
18:     return batchTests = {t_1, t_2, ..., t_VN_p}
19: end procedure

20: procedure CliqueSampling(SG)
21:     constraints cns = true, P = SG.V
22:     while P is not empty do
23:         randomly pick and remove a vertex v from P
24:         if satisfiabile(cns ∧ (v.le == v.rv)) then
25:             cns = cns ∧ (v.le == v.rv)
26:             P = P ∩ SG.E(v)
27:         else if cns has one constraint u.le == u.rv then
28:             mutex.lock()    // Protect shared graphs
29:             SG.E(v) = SG.E(v)\{u}
30:             SG.E(u) = SG.E(u)\{v}
31:             mutex.unlock()
32:         end if
33:     end while
34:     Use SMT solver to solve cns and return the test
35: end procedure
```

while the average size is 20 in the small clique scenario (Figure 7-7(b)). In the large clique scenario, the 8-trigger condition is more likely to be in the overlap areas of many maximal

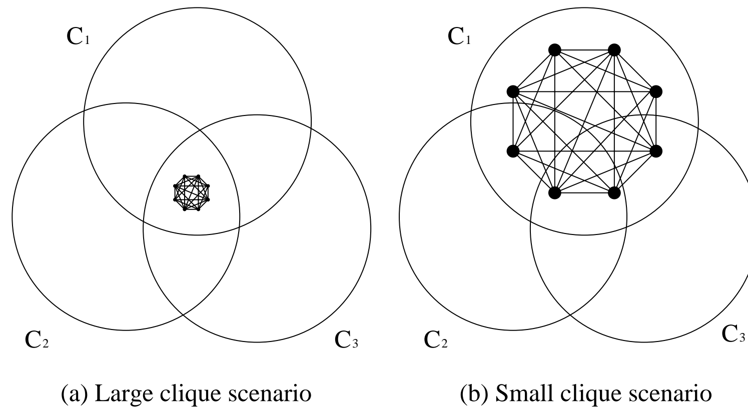


Figure 7-7. The relative size of trigger conditions compared to maximal SAT cliques.

satisfiable cliques as shown in Figure 7-7(a). In this case, random sampling (Algorithm 7) can easily activate the trigger condition by generating a test vector to cover any of the maximal satisfiable cliques that are a super set of the trigger condition. On the other hand, the size of the 8-trigger condition is close to the average maximal clique size in the small clique scenario. As a result, it is less likely to be activated by random sampling since it is covered by a small number of maximal satisfiable cliques. In the extreme case, e.g., the size of trigger condition is the same as the size of maximal SAT cliques, we need to enumerate all maximal SAT cliques as Algorithm 6. In fact, it is the best any test generation approach for this case. In most of the benchmarks, we observe a relatively large maximal satisfiable cliques compared to trigger points, as shown in Section 7.3.6.

In summary, our paradigm reduced and mapped the problem of trigger activation to the problem of covering maximal satisfiable cliques. The choice between clique enumeration and random sampling is based on the relative size of maximal satisfiable cliques and the trigger points. When an adversary is allowed to construct any size of trigger condition, e.g., a size close to the maximal SAT cliques, Algorithm 6 is the optimum way to generate tests. However, it is not realistic in practice. An adversary tends to select a small number of trigger points considering area and power constraints in the design and to bypass side-channel analysis. In this scenario, random sampling (Algorithm 7) further reduces the problem size by selecting the representative maximal satisfiable cliques. As shown in the above example, each 8-trigger

condition could possibly be covered by a large number of maximal satisfiable cliques of average size 200. If one of them is sampled by our algorithm, the trigger activation problem is solved. It also points out an interesting direction to improve TARMAC. Instead of randomly sampling each time, a biased sampling technique could be beneficial to instruct the sampling process to cover cliques that have less overlap with already covered ones. In order to improve the performance further, we have modified TARMAC such that multiple threads can perform clique sampling in parallel. This will enable an efficient and scalable test generation framework for activating rare triggers.

7.3 Experiments

7.3.1 Experimental Setup

The TARMAC framework is implemented in C++ with Z3 [121] as our SMT solver. This framework first parses gate-level Verilog files into design graphs ($\mathcal{DG}$). Then, for each signal in PTS, we utilizes Z3 C++ API to compute its logic expression. For sequential circuits, all registers are treated as free variables with the assumption of full scan mode. Next, this framework constructs a satisfiability graph ($\mathcal{SG}$) and continuously samples maximal satisfiability cliques ($\mathcal{MSC}$) as shown in Algorithm 7. For each sampled $\mathcal{MSC}$, function call to Z3 is used to produce a test. For multithreading (TARMAC$_p$) in Algorithm 8, C++ pthread library is used to create different threads.

We conducted a wide variety of experiments on a server with Intel Xeon E5-2698 CPU @2.20GHz to evaluate the performance of TARMAC compared to random test vectors and $N-$detect approach (MERO [12]). In this chapter, we used the same benchmarks (ISCAS-85 [30] and ISCAS-89 [56]) from [12] to enable a fair comparison with MERO. We have also used two large designs (memory controller from TrustHub [105] and MIPS processor from OpenCores [96], MEM and MIPS for short) to demonstrate the scalability of our approach. The experimental setup is shown in Figure 7-8. We first ran a number of random simulations (100K for ISCAS and one million for MEM and MIPS) and computed the probability of each signal. Rareness threshold is set to 0.1 for ISCAS benchmarks and 0.005 for the other designs.

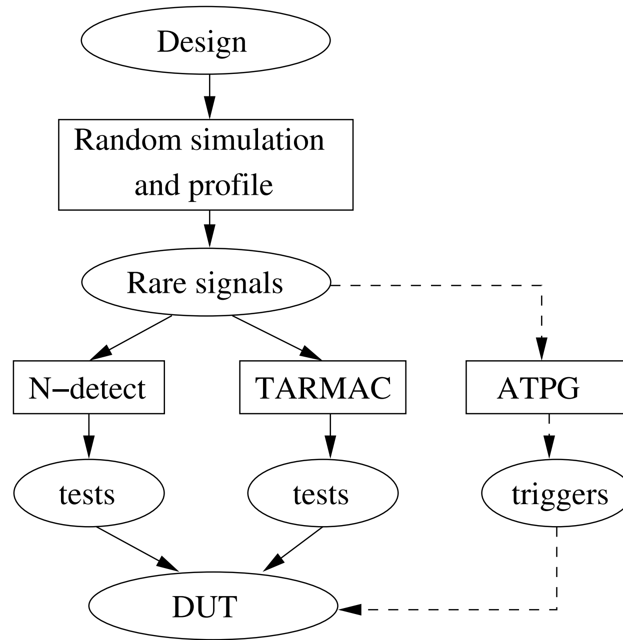


Figure 7-8. Experimental setup for evaluation of TARMAC compared to $N-$detect approach.

For each benchmark, 1000 trigger conditions were randomly sampled and validated using ATPG. After sampling 1000 valid trigger conditions, each of them was individually integrated into the original design to construct a design under test (DUT). In other words, there are 1000 DUTs from each benchmark with one trigger condition for evaluation. We applied $N-$detect approach (MERO [12]), TARMAC (Algorithm 7) and TARMAC$_p$ (Algorithm 8) to generate the test sets with the rare signals as potential trigger signals (PTS). Finally, we applied test sets to each DUT and collected trigger condition coverage. For all experiments, we fixed $N = 1000$ for $N-$detect approaches.

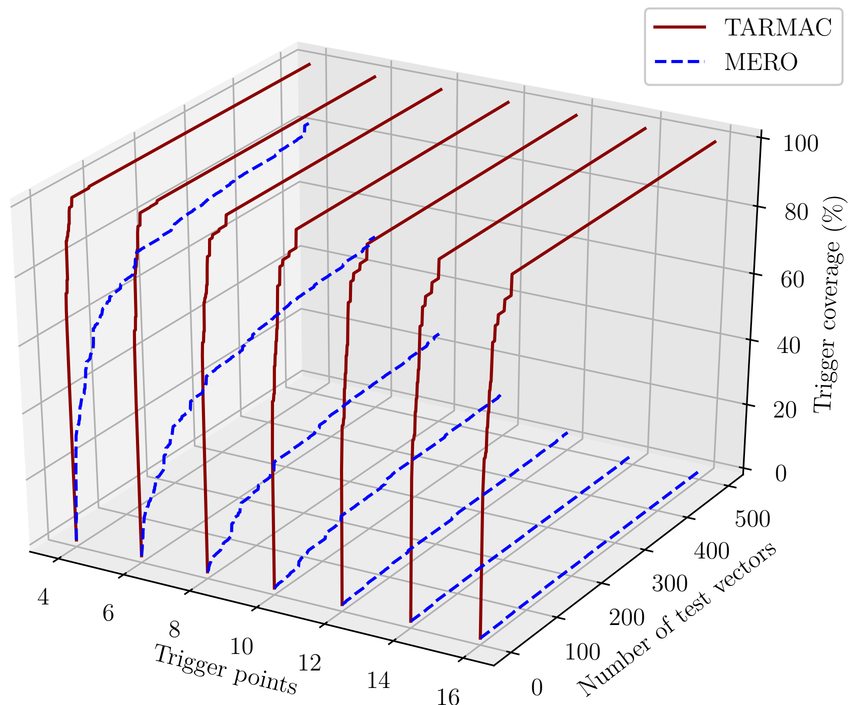
7.3.2 The Effects of Trigger Points

In the first experiment, we wanted to explore the effects of trigger points on the trigger coverage of MERO and TARMAC. When a trigger condition has less trigger points (e.g., 4), it has higher probability to be activated by random simulation. One the other hand, a trigger condition with more rare signals is much harder to activate. For example, the probability of activating a 16-trigger condition is less than 10^{-16} when these signals are independent and rareness threshold is 0.1.

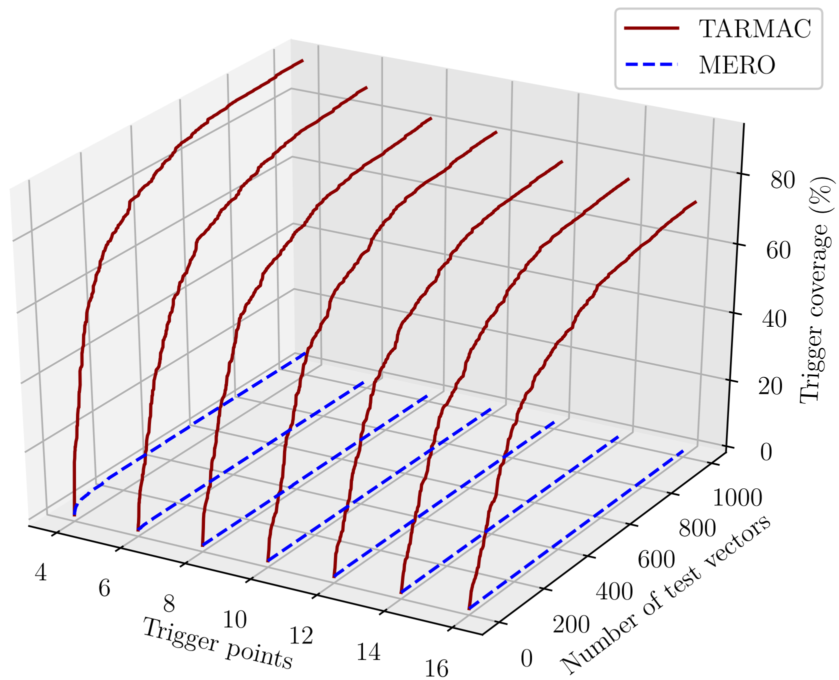
We evaluated both MERO and TARMAC on c2670 and MIPS, with various number of trigger points between 4 and 16. The results of TARMAC and MERO are shown in Figure 7-9. Each line represents trigger condition coverage with respect to the number of test vectors applied to DUTs with a fixed number of trigger points. As the results suggest, the performance of MERO deteriorated sharply with increasing trigger points, while TARMAC maintained high coverage for both benchmarks. For small number of trigger points (e.g., 4), MERO can achieve good coverage in c2670. However, its coverage for large number of trigger points (e.g., 16) is extremely poor with less than 5% coverage. On the other hand, TARMAC can achieve 100% coverage with less than 100 test vectors even for 16-trigger conditions. As 16-trigger condition is more rare than 4-trigger ones, TARMAC took more test vectors to achieve the same coverage in MIPS as shown in Figure 7-9B. Therefore, TARMAC is more resilient to the increasing number of trigger points and good at activating extremely rare-to-activate trigger conditions. In the remaining experiments, we fix the number of trigger points to be 8 since it is a common number of trigger points in TrustHub [105] and it allows MERO to achieve a reasonable trigger condition coverage for comparison.

7.3.3 Performance Evaluation

In this experiment, we compared the trigger condition coverage of TARMAC to random approach and MERO over all benchmarks. To get a fair comparison of trigger coverage, we evaluated the trigger coverage with the same number of test vectors. Note that the length of MERO test vectors cannot be controlled arbitrarily since it depends on the $N-$detect criteria and the number of initial random vectors R. Hence, we first ran MERO with ($R = 100,000, N = 1000$) for ISCAS benchmarks as suggested in [12] and ($R = 100,000, N = 1000$) for two large benchmarks. After MERO finished, we ran TARMAC to generate the same number of test vectors as MERO for each benchmark. The trigger coverage comparison of TARMAC with random and MERO test vectors is shown in Table 7-1.



A c2670 [30]



B MIPS Processor [96]

Figure 7-9. Trigger condition coverage of TARMAC and MERO on c2670 and MIPS with respect to the number of test vectors and the number of trigger points.

Table 7-1. Comparison of TARMAC with random simulation and MERO for trigger activation coverage over 1000 randomly sampled 8-trigger conditions.

	No.	Random		MERO [12]			TARMAC					TARMAC$_p$(64)	
	of rare	Test	Cov.	Test	Cov.	Time	Test	Cov.	Imp./	Imp./	Time	Time	Imp./
Bench	signals	Len	(%)	Len	(%)	(s)	Len	(%)	Rand	MERO	(s)	(s)	MERO
c2670	43	100K	0.3	6820	38.2	1268	6820	100	333x	2.6x	257	4.3	295x
c5315	164	100K	1.1	9232	50.6	4396	9232	98.8	89.8x	1.9x	682	13.3	330x
c6288	169	100K	18.9	5044	76.6	596	5044	95.0	5.0x	1.2x	638	10.0	60x
c7552	278	100K	0	14914	5.6	7871	14914	66.5	∞	11.9x	2185	41.6	189x
s13207	604	100K	0	44534	1.9	15047	44534	94.4	∞	49.7x	5417	105.3	143x
s15850	649	100K	0	39101	3	17000	39101	88.7	∞	29.6x	11337	205.4	83x
s35932	1152	100K	100	4047	100	49616	4047	100	1x	1x	1947	38.9	1275x
MEM	1306	1M	0	28542	0	89747	28542	98.6	∞	∞	15753	330.5	271x
MIPS	906	1M	0	25042	0.2	273807	25042	95.6	∞	472x	19458	391.9	699x
average	586	300K	13.4	19697	30.7	51039	19697	93.1	>107x	>71x	6408	126.8	402x

Table 7-2. Comparison of TARMAC with random simulation and MERO for trigger activation coverage over 1000 randomly sampled 8-trigger conditions.

	MERO			TARMAC				
bench	Test Length	Cov. (%)	Time (s)	Test Length	Reduction	Cov. (%)	Time (s)	Improvement
c2670	6820	38.2	1268	1	6820x	51.4	0.05	25360x
c5315	9232	50.6	4396	217	42.5x	50.6	19.1	230x
c6288	5044	76.6	596	284	17.8x	76.6	34.8	17x
c7552	14914	5.6	7871	175	85.2x	5.6	31.2	252x
s13207	44534	1.9	15047	5	8907x	2.6	0.8	18809x
s15850	39101	3	17000	13	3008x	3.3	4.3	3953x
MEM	28542	0	89747	1	28542x	1.9	1.1	81588x
MIPS	25042	0.2	273807	1	25042x	0.8	1.8	152115x
average	21653	22.0	51216	87	249x	24.1	11.6	4415x

From Table 7-1, we can see that TARMAC can achieve several orders-of-magnitude trigger coverage improvement over random test vectors in ISCAS benchmarks. TARMAC can provide and up to 49 times improvement in trigger coverage over MERO with four times reduction for generation of the same number of test vectors in the ISCAS benchmarks. For most benchmarks, TARMAC covered over 90% of the trigger conditions, while random and MERO test vectors missed most of them. In small benchmarks, such as c2670, c5315 and c6288, MERO outperformed random test vectors and achieved reasonable trigger condition coverage. However, in large benchmarks such as c7552, s13207 and s15850, the performance of MERO is very poor, with less than 6% trigger coverage. TARMAC, on the other hand, outperformed MERO in all ISCAS benchmarks with 91.9% trigger coverage on average. With the same number of test vectors, TARMAC can cover the extremely hard-to-activate trigger conditions that are left after applying both random test vectors and MERO with significantly less effort.

It is interesting to find that all three approaches did a great job in covering all trigger conditions in s35932. One of the reasons is that a lot of rare signals in s35932 can be satisfied together as shown in Section 7.3.6. Another observation is that the quality of MERO is partially dependent on the quality of random test vectors. For example, with 18.9% and 100% trigger activation coverage from random test vectors for c6288 and s35932, respectively, test

vectors from MERO can cover 76.6% and 100%. However, for benchmarks such as c7552 and s31207, test vectors of MERO can only achieve trigger coverage of 5.6% and 1.9%, respectively, since random test vectors cannot cover any trigger conditions. The limited improvement from random test vectors to MERO is due to the simple bit flipping to search for good vectors in MERO.

For the two large benchmark, MEM and MIPS, the number of rare signals are in the order of 1000. Since each trigger condition contains 8 rare signals with rareness threshold of 0.005, the probability of trigger conditions is less than 10^{-18}. It is expected that one million random simulations could not achieve good coverage. The test vectors generated by MERO also achieved poor coverage, 0% in memory controller, and 0.2% in MIPS. On the other hand, TARMAC is able to cover majority of the trigger conditions efficiently. For example, TARMAC covered 95.6% of trigger conditions in MIPS using the same amount of test vectors as MERO, but finished generation in 6 hours. Note that the average test generation of TARMAC for one test vector is less than one second. This demonstrates that TARMAC is scalable for large designs, while MERO is not suitable for large designs.

Overall, TARMAC provides drastic improvement in both trigger coverage (more than 107x and 71x over random test vectors and MERO, respectively) and test generation time (8x).

7.3.4 Parallelism Evaluation

In this experiment, we run Algorithm 8 with 64 threads in parallel to generate the same amount of test vectors as MERO. The results are shown in the last two columns of Table 7-1. Since the overall coverage of TARMAC and TARMAC$_p$ are similar, the coverage of TARMAC$_p$ is omitted. Overall, TARMAC$_p$ with 64 threads can achieve up to 1275x (402x on average) improvement over MERO.

To evaluate the utilization of multi-core architectures, we applied Algorithm 8 with different number of threads in MIPS. The result is shown in Figure 7-10. As we can see, the utilization of multiple cores is very high. Compared to the single thread scenario, 64 threads in parallel can achieve 49 times speedup. In other words, the overhead of protecting shared

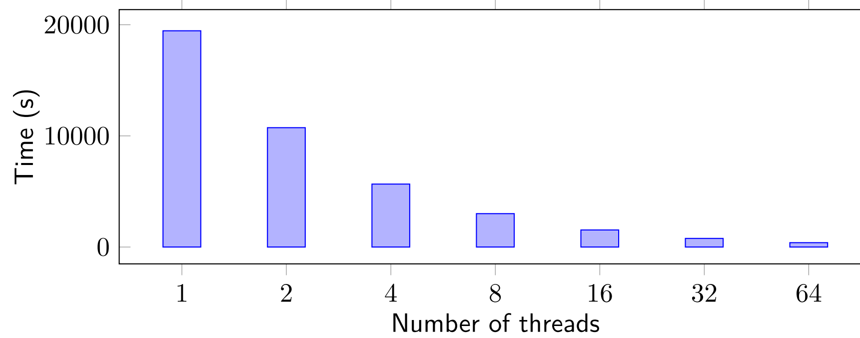


Figure 7-10. The time of Algorithm 8 applied in MIPS with different number of threads.

satisfiability graph using mutex in Algorithm 8 is negligible. As we can see, Algorithm 8 can generate 25042 test vectors for MIPS in approximately 6.5 minutes.

7.3.5 Compactness and Efficiency

To compare the compactness and efficiency of TARMAC with MERO, we terminated TARMAC when it just surpassed the same trigger coverage as MERO. In this experiment, we omit the benchmarks s35932 that MERO achieved full coverage, because 100% coverage can be achieved with much fewer test vectors but test length is not a configurable parameter in MERO. It would be an unfair comparison if we compare the test length of TARMAC to the number of s35932 in Table 7-1. The results of the remaining benchmarks are shown in Table 7-2. Note that one test vector in TARMAC can outperform the trigger coverage of MERO for c2670, MEM and MIPS. In all the other benchmarks, the difference of corresponding trigger coverage is minimal.

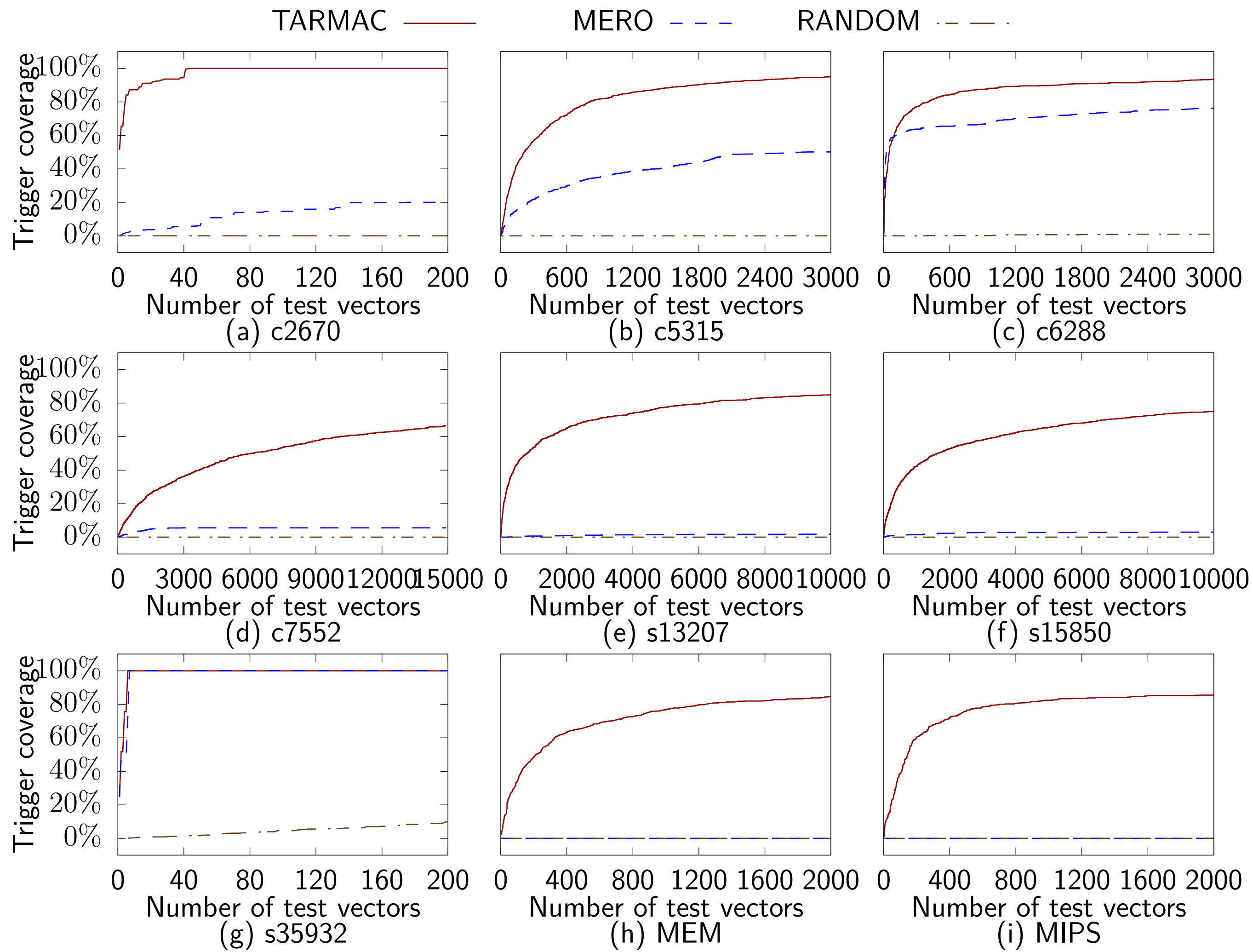


Figure 7-11. Trigger coverage with respect to the number of test vectors.

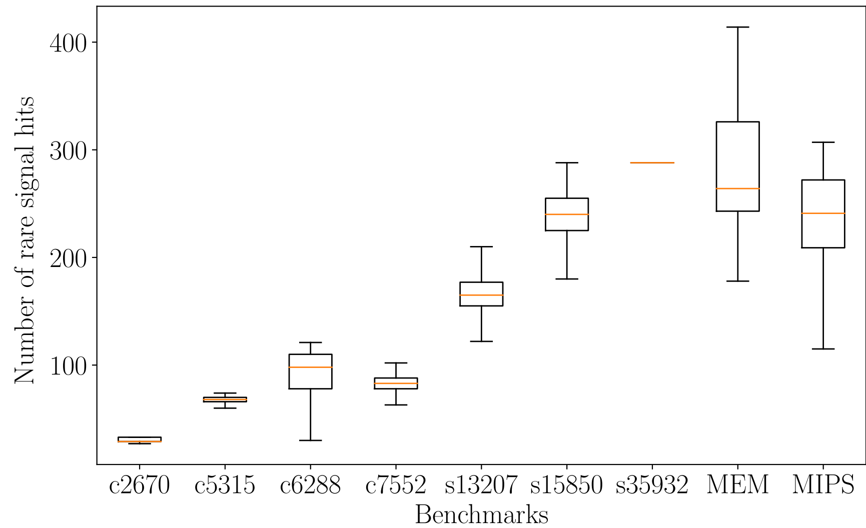
Table 7-2 suggests that test vectors generated by TARMAC are several orders-of-magnitude more compact than MERO. For ISCAS benchmarks, the average reduction of test vectors is in the order of hundreds to achieve the same coverage. The compactness gap becomes larger and larger when the size of design grows. For example, while most of the reductions in small benchmarks (combinational circuits) are within 100 times, the reductions in sequential benchmarks grows to the order of thousands. In MEM and MIPS, the reduction even goes beyond 25 thousands.

The improvement in test generation time follows the same trend as test length reduction. For example, while most of the time improvements in small benchmarks are within the order of hundreds, the improvements in sequential benchmarks grows to the order of thousands even ten thousands. Finally, the improvement in MIPS processor even goes beyond 152 thousands.

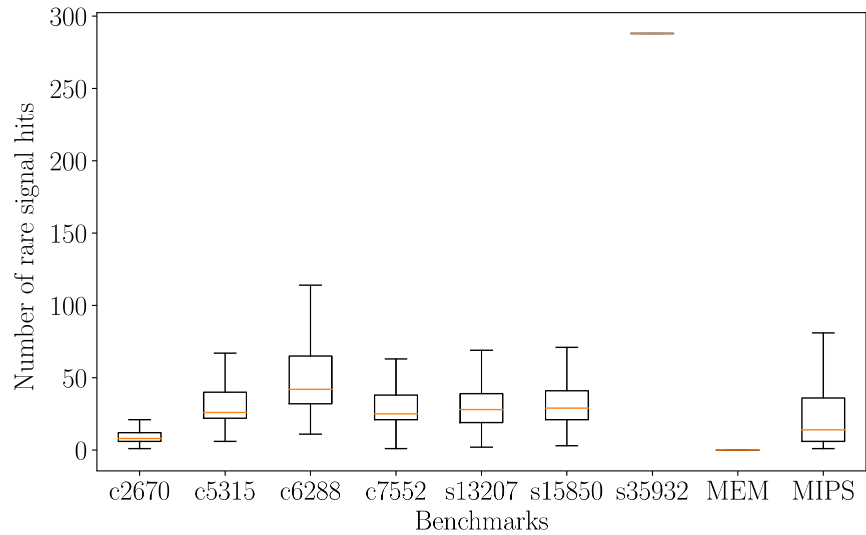
From the perspective of debug engineer, efficiency of a test generation approach consists of two aspects. The first one is test generation time. From Table 7-2, we can see that the improvements of test generation time over MERO are several orders of magnitude. The other one is test length as it decides how many simulations or emulations are needed, which dominates debug time. As a result, a compact test set can lead to significant reduction in overall validation effort. Combining both improvements of test generation and reduction of test length as shown in Table 7-2, the efficiency of TARMAC is several orders of magnitude better than MERO.

7.3.6 Trigger Coverage

For better illustration of trigger coverage, we ran all benchmarks long enough and plotted the trigger coverage with respect to the number of test vectors in Figure 7-11. The x-axis represents the number of tests applied to DUTs, and the y-axis represents the percentage of activated trigger conditions. The efficiency in trigger coverage is the gradient of trigger coverage curves. In most of the figures, TARMAC has much steeper slopes than MERO and the curves of random approach are almost flat. The results demonstrated that TARMAC can cover more trigger conditions faster (with significantly less test vectors) than MERO for most



A TARMAC (Our Approach)



B MERO [12]

Figure 7-12. The distribution of rare signal hits by the generated test set in all benchmarks.

of the benchmarks. For example, with 200 test vectors in c2670, TARMAC already activated all the trigger conditions, while MERO only achieved 20% coverage.

These figures reveal that each vector in TARMAC is able to activate more potential trigger conditions than MERO. As stated in Lemma 4, each test vector can cover all the subgraphs of a satisfiable clique. Hence, if one test vector can activate more rare signals, it covers a larger clique and likely to activate more potential trigger conditions. Therefore, we

define the *quality of a test vector* as the number of rare signals that it can cover (activate). To validate whether the quality of a test vector is the reason for different trigger coverage efficiency, we counted the number of rare signals satisfying their rare values (*rare signal hits*, for short) for each test vector. Figure 7-12 shows the distribution of rare signal hits by each test vector. The results show that the numbers of rare signal hits are significantly larger in TARMAC (except for the comparable numbers in c6288 and s35932), which is consistent with observations in Figure 7-11 considering the coverage of trigger conditions. From Algorithm 7, the number of rare signal hits is the same as the size of each sampled maximal satisfiable clique in TARMAC. While in MERO, the number of rare signal hits is the best number of hits after one round of bit flipping from a random test vector. Clearly, the rare signal hits from MERO should be statistically always lower than TARMAC as the rare signal hits in TARMAC are optimal. Moreover, the quality of test vectors in MERO is not guaranteed, since it partially depends on the initial random vectors. As a result, MERO has low rare signal hits (normally less than 50), which is significantly smaller than rare signal hits in TARMAC.

7.4 Summary

Trigger activation is a fundamental challenge in detection of hardware Trojans. While prior efforts using statistical test generation are promising, they are neither scalable for large designs nor suitable for activating extremely rare trigger conditions in stealthy Trojans. In this chapter, we introduced a new paradigm to solve trigger activation problem. This chapter made the following important contributions. 1) Our approach is the first attempt in mapping the problem of test generation for trigger activation to the problem of covering maximal satisfiability cliques. 2) We proved that valid trigger conditions and satisfiability cliques are one-to-one mapping. We also proved that the test vectors generated by our paradigm are both complete and compact. 3) We presented efficient test generation algorithms to repeatedly sample maximal satisfiability cliques and generate a test vector for each of them. We explored the effectiveness of random sampling, lazy construction as well as multi-threading to improve the test generation efficiency. Our experimental results demonstrated that our approach is both

scalable and effective in generating efficient test vectors for a wide variety of trigger conditions. Our approach outperforms the state-of-the-art techniques by several orders-of-magnitude in terms of trigger coverage, test length as well as test generation time. Our test generation algorithms can be utilized for activating extremely rare trigger conditions to fulfill diverse requirements such as improvement of functional (trigger) coverage as well as side-channel sensitivity.

CHAPTER 8
TROJAN DETECTION USING CURRENT-BASED SIDE-CHANNEL ANALYSIS

Detection of hardware Trojans is vital to ensure the security and trustworthiness of System-on-Chip (SoC) designs. Side-channel analysis is effective for Trojan detection by analyzing various side-channel signatures such as power, current and delay. In this chapter, we target the dynamic current as our side-channel signature and propose an efficient test generation technique to maximize the side-channel sensitivity for Trojan detection. Note that this approach can also be extended to the other side-channel parameters with suitable modifications of the evaluation criterion. While early work on current-aware test generation has proposed several promising ideas, there are two major challenges in applying it on large designs: (i) the test generation time grows exponentially with the design complexity, and (ii) it is infeasible to detect Trojans since the side-channel sensitivity is marginal compared to the noise and process variations.

Our proposed work, referred to as MaxSense, addresses both challenges by effectively exploiting the affinity between the inputs and rare (suspicious) nodes. We formalize the test generation problem as a searching problem and solve the optimization using genetic algorithm. The basic idea is to quickly find profitable ordered pairs of test patterns, $T = \{t_i\} = \{<u_i,v_i>\}$, that can maximize switching in the suspicious regions while minimize switching in the rest of the circuit. There are two objectives of MaxSense shown in Figure 8-1. We refer t_i as an ordered pair of test patterns with u_i as its first pattern and v_i as its second pattern. The first pattern in the ordered pair (u_i) tries to maximize the activation of the suspicious regions. The second pattern (v_i) needs to simultaneously satisfy two objectives. The first objective is to maximize switching in the suspicious regions, similar to Chapter 7. The second objective is to minimize the switching in the rest of the design, such that the side-channel sensitivity is maximized. As demonstrated in Section 8.1.3, the selections of both u_i and v_i are equally important to enable efficient test generation for effective side-channel analysis. Specifically, this chapter makes the following major contributions:

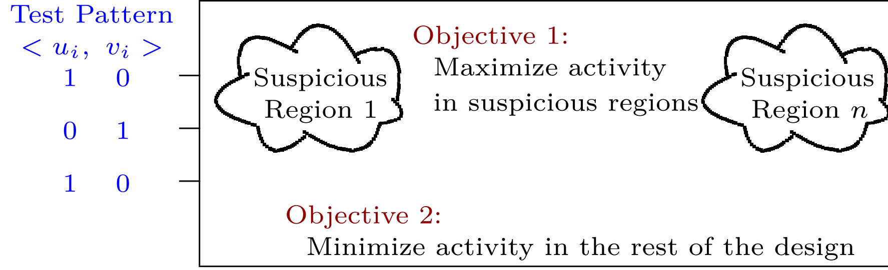


Figure 8-1. The two objectives to maximize the sensitivity in current-based side-channel analysis.

- Exploits the input affinity to identify test patterns that can maximize switching in the suspicious (target) regions while minimizing switching in the rest of the circuit in order to significantly improve the side-channel sensitivity.
- Proposes a fast and effective Satisfiability Modulo Theories (SMT) based approach to increase the activation probability in the suspicious regions.
- Utilizes genetic algorithm to quickly find the profitable test patterns by exploiting affinity of the inputs to the suspicious regions, thus improving the side-channel sensitivity.
- The significant improvement in sensitivity enabled MaxSense to detect the majority of Trojans (out of randomly inserted 1000 Trojans), while the state-of-the-art approaches can detect less than 2% Trojans.

The chapter is organized as follows. Section 8.1 provides problem formulation and motivates the need for our work. Section 8.2 describes our test generation framework. Section 8.3 presents experimental results. Finally, Section 8.4 concludes the chapter.

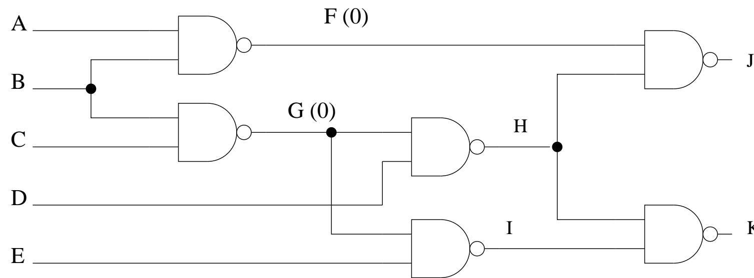
8.1 Problem Formulation and Motivation

8.1.1 Problem Formulation

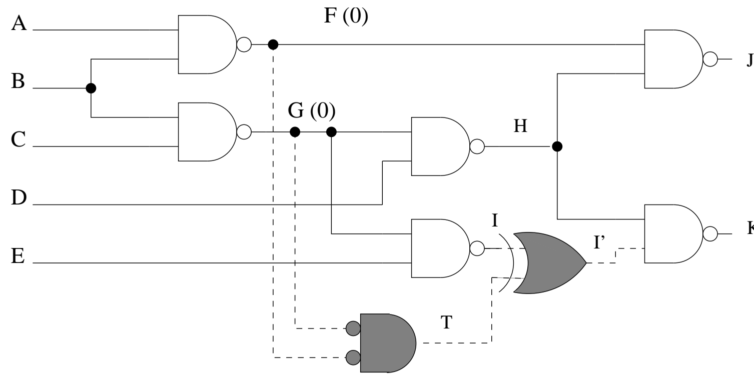
Our goal is to generate l compact ordered pairs of test patterns $\{<u_i,v_i>\}$ $(i = 1, 2, ..., l)$ that can maximize the dynamic current based side-channel sensitivity. For each pair of the test patterns $<u_i,v_i>$, the current switching in the golden design G (called the original switching) is measured by applying the first pattern u_i followed by the second pattern v_i, denoted $switch^{G}_{u_i,v_i}$. The current switching in the Trojan-inserted design G^T is defined in the same way, i.e., $switch^{GT}_{u_i,v_i}$. The *relative switching* is computed as $|switch^{G}_{u_i,v_i} - switch^{GT}_{u_i,v_i}|/switch^{G}_{u_i,v_i}$.

The ***sensitivity*** of a Trojan T is defined as the maximum of the relative switching over all test patterns, as shown in Equation 8-1.

$$sensitivity_T = max_{(i=1,2,\ldots,l)}(\frac{|switch^G_{u_i,v_i} - switch^{G^T}_{u_i,v_i}|}{switch^G_{u_i,v_i}}) \tag{8-1}$$



A The netlist of c17 from ISCAS-85 [30].



B The netlist of c17 with a Trojan inserted.

Figure 8-2. The example of a Trojan inserted into c17.

8.1.2 An Illustrative Example

To illustrate how to improve the sensitivity in dynamic current based side-channel analysis, we first use a small benchmark c17 from ISCAS-85 [30] as an example with its netlist shown in Figure 8-2A. We set *rareness threshold* to be 0.3, i.e., rare nodes are defined as the signals whose rare values are satisfied with less than 30% probability in random simulations. For example, F and G are two rare nodes with rare value 0 in Figure 8-2A.

Assume an attacker uses rare nodes F and G to construct the trigger condition and the Trojan is shown using dashed lines in Figure 8-2B. For this small design, we enumerate all possible pairs of test patterns and compute each sensitivity. The best pair of test patterns

is $<u,v> = <11100,10100>$ on inputs (<A,B,C,D,E>), with only B switches from '1' to '0'. The current switching in the golden design $switch_{u,v}^{G}$ is 4 (switching of signals B, F, G, and J) and the current switching in the Trojan inserted design $switch_{u,v}^{G^t}$ is 7 (switching of signals B, F, G, J, T, I', and K). Thus the sensitivity is 75% in this example. *An important observation is that by flipping only a small number of relevant inputs (B in this example) while preserving the others, the switching activities in the Trojan area are maximized while the current switching in the golden design is minimized.* In other words, if we can exploit the ***affinity*** between inputs and the rare nodes while creating a pair of test patterns (u followed by v), it can lead to a significant improvement in sensitivity for Trojan detection. Experimental results in Section 8.3.4 (Figure 8-8) demonstrates that affinity is useful in practice.

8.1.3 Motivation and Research Challenges

By inspecting the capability of $<11100,10100>$ for c17, we want to divide the task of searching for effective pairs of test patterns into two sub-problems. (1) Generation of *the first pattern that tries to maximize activation of rare nodes with their respective rare values*, e.g., 11100 in the previous example. As the difference of current switching in designs with/without Trojans comes from the switching of the inserted circuits, the sensitivity can be improved if the switching activity is maximized in these suspicious regions. (2) Given the first pattern u generated in the previous step, searching for the most profitable *second pattern v which is responsible for both maximization of switching in rare nodes and minimization of switching in non-rare nodes*, e.g., 10100 in the previous example.

However, there are three main challenges in searching for effective pairs of test patterns.

1. Randomly selected pairs may not lead to high sensitivity, even if the two patterns are similar. For example, if we apply $<u,v> = <11100,10100>$ to the previous example, the switching activities in G and G^T are the same, revealing no side-channel footprint.

2. The whole search space is exponentially large (2^n, where n is the number of inputs in the design). So, searching for the whole space is not feasible. Based on affinity heuristic, the neighbor of u within a small Hamming distance (e.g., less than k) is the optimized search space. One naive way is to use breadth-first-search (BFS) according to the Hamming distance. However, the searching complexity is still $O(n^k)$.

3. There is a tradeoff between introducing switching in the rare nodes and minimizing switching in the golden design. We need to introduce a reasonably large switching in rare nodes, since we have no knowledge of the trigger condition. However, for a design with thousands of rare nodes, introducing switching for all of them can lead to a significant increase in the switching of the golden design. In that case, even if the Trojan is fully activated, the sensitivity (tens of the extra switching divided by thousands of the original switching) can be too small compared to the process and noise margins.

Our approach addresses these challenges by using an SMT-based first pattern generation to maximize the activation of rare nodes and using genetic algorithm as an approximate and optimized replacement of BFS to search for the most profitable second patterns. Based on input affinity, we initialize GA with random test patterns that have fixed small Hamming distance from the first pattern. By crossover and mutation, the Hamming distance is expected to grow slowly. After several generations, the majority of the profitable test patterns in the expected search space are likely to be visited.

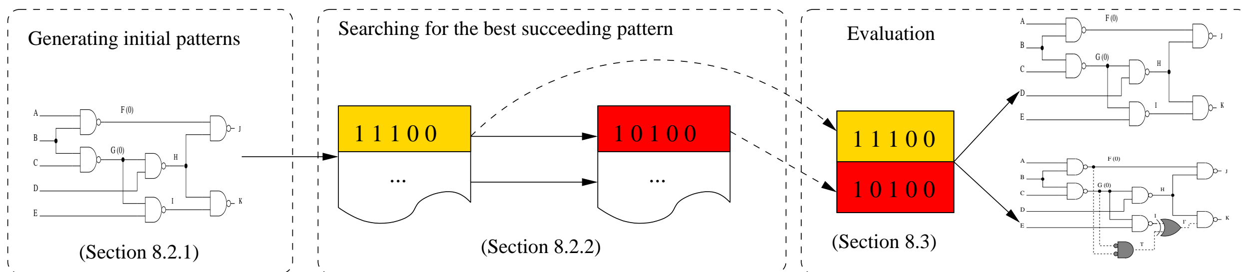


Figure 8-3. The overview of our framework MaxSense.

8.2 Generation of Effective Test Patterns

Figure 8-3 shows an overview of our proposed approach (MaxSense). It has three important steps. The first step generates the first patterns to maximize activation of rare nodes with their respective rare values (Section 8.2.1). The next step finds the most profitable second patterns for both maximization of switching in rare nodes and minimization of switching in non-rare nodes (Section 8.2.2). Finally, we evaluate the quality of the generated pairs of test patterns (Section 8.3).

8.2.1 Generation of the First Patterns

The sensitivity of side-channel analysis is maximized if the ordered pairs of test patterns are able to maximize activation of rare nodes with their respective rare values, i.e., partially or fully activate trigger conditions. The basic idea is to increase the activities of rare signals to increase the probability of activating the unknown trigger conditions.

The test patterns generated by N-detect approach [12] is promising to achieve this goal. However, efficiency is the main bottleneck of the N-detect approach. The N-detect approach requires one simulation in each bit flipping of a single initial random test pattern. Therefore, if the number of initial random test vectors are large and the design is complex, it takes long time to finish all simulations. What is worse, the N-detect approach cannot run in parallel. It is due to the fact that the N-detect criterion relies on the overall performance of all test vectors, and we cannot evaluate the quality of a single test pattern without evaluating all the other patterns. For example, if a set of test patterns already cover all but one rare signal N times, a new test pattern that is able to cover the remaining rare signal is better than a test vector that is able to cover hundreds of rare signals that are already activated by N times. To address these inherent problems of N−detect approach, we propose an effective and parallelizable test generation approach utilizing an SMT solver to produce the first patterns.

Before describing the details of our SMT-based approach, we first introduce logic expressions for signals that will be used in our approach. For each signal, we define its logic expression as an expression that is composed by controllable signals, such as primary inputs, flip-flops with a scan chain. For example, the logic expression for signal F is $!(A \wedge B)$ in Figure 8-2. This logic expression is stored in $F.expr$ and the rare value of F is stored in $F.rv$.

Algorithm 9 shows our SMT-based approach. The main part of Algorithm 9 contains M iterations, each of which generates one test pattern by calling an SMT solver. In each iteration, we first generate a random permutation of all rare signals in RP and initialize an SMT expression S to be true. Then, we construct an expression $e_i = (rp_i.expr == rp_i.rv)$ for each rare signal $rp_i \in RP$. These expressions from rare signals in RP are added one by one

to our SMT expression S. To maintain S to be satisfiable, we skip every rare signal rp_i when $S \wedge e_i$ is unsatisfiable. In other words, S contains rare signals that can construct a valid trigger condition. When we have "enough" ($TriggerLimit$) satisfiable rare signals in S, we get an input pattern by solving S using an SMT solver. By choosing RP as a random permutation of all rare signals in each iteration, Algorithm 9 tries to generate test patterns that can activate different combinations of rare signals. At the end of Algorithm 9, M test patterns are returned as the set of first patterns, each of which is able to activate a combination of rare signals.

The performance of Algorithm 9 depends on the total number of test patterns and the complexity of the design. The number of iterations is controlled by the user-defined parameter M, which is the size of test patterns. In each iteration, one random permutation of rare signals RS is performed with time complexity $O(|RS|)$. In the inner loop (Line 8-17), we are looking for no more than $TriggerLimit$ rare signals that can be satisfied together. Therefore, the number of iterations of the inner loop is between $TriggerLimit$ and $|RS|$. In the worst case, all rare signals are tried and cannot find more than $TriggerLimit$ satisfiable rare signals. Therefore, the worst-case run time of Algorithm 9 is $O(M \times |RS| \times T(SMT))$, where $T(SMT)$ is the worst time to solve at most $TriggerLimit$ expressions by an SMT solver. The running time can be reduced with multi-core architectures. It is easy to see that Line 5-18 can be run in parallel. In other words, if we want to generate M test vectors and we have C cores, each core can generate M/C test vectors, i.e., C times speedup.

8.2.2 Searching for the Best Succeeding Pattern

The second task is to find the best succeeding pattern v_i for each u_i (identified in Section 8.2.1), such that the relative switching is maximized. There are many selection algorithms in the literature, including genetic algorithm, simulated annealing and machine learning. While all of them provided promising results, we used genetic algorithm in our framework primarily due to its effectiveness in exploiting affinity, and delivering profitable second patterns in a small number of iterations as described in Section 8.3.4.

Algorithm 9 First Pattern Generation using SMT Solver

```
1: procedure SMT(circuit netlist, rare signals RS, the number of test vectors M)
2:     initialize first pattern set FP = ∅
3:     compute logic expression for each rare signal
4:     for k = 1 to M do
5:         RP = random_permutation(RS)
6:         initialize SMT expression S = 1
7:         the total number of satisfiable expression total = 0
8:         for each rare signal rp_i ∈ RP do
9:             new expression e_i = (rp_i.expr == rp_i.rv)
10:            if satisfiable(S ∧ e_i) then
11:                S = S ∧ e_i
12:                total = total + 1
13:            end if
14:            if total > TriggerLimit then
15:                break
16:            end if
17:        end for
18:        solve S and get input pattern u_k
19:        FP = FP ∪ {u_k}
20:    end for
21:    return FP
22: end procedure
```

Genetic algorithm forms the main part of Algorithm 10, which consists of four major steps: initialization, fitness computation, selection, and crossover and mutation. The ***fitness*** is defined in Equation 8-2, where $rare_switch^{G}_{u,v}$ represents the current switching of all rare nodes in G when applying the test pattern u followed by v. A profitable test pattern should maximize the current switching in rare nodes to increase the probability of activating a Trojan, and minimize the total switching in the golden design. The best second pattern v_i for a given preceding u_i is the one achieving the highest fitness value over all generations (line 11). The first iteration of GA for c17 is shown in Figure 8-4, assuming 4 individuals in each generation.

$$fitness_u(v) = \frac{rare_switch^{G}_{u,v}}{switch^{G}_{u,v}} \tag{8-2}$$

Algorithm 10 Second Pattern Generation using GA

```
1:  procedure TestGeneration(circuit netlist, rare signals RS, first patterns FP = {u_i})
2:      for each first pattern u_i ∈ FP do
3:          Initialization of GA with u_i
4:          For each individual v, compute fitness_{u_i}(v) by simulating the netlist with the pair
    of test patterns (u_i, v)
5:          for gen = 1 to generations do
6:              Selection of parents from the gen^th generation based on fitness values
7:              Single point crossover to produce children
8:              Single point mutation according to mutation rate
9:              Compute fitness for the children ((gen + 1)^th generation)
10:         end for
11:         Select the best individual over all generations as v_i
12:     end for
13:     return the pairs of test patterns {<u_i,v_i>}
14: end procedure
```

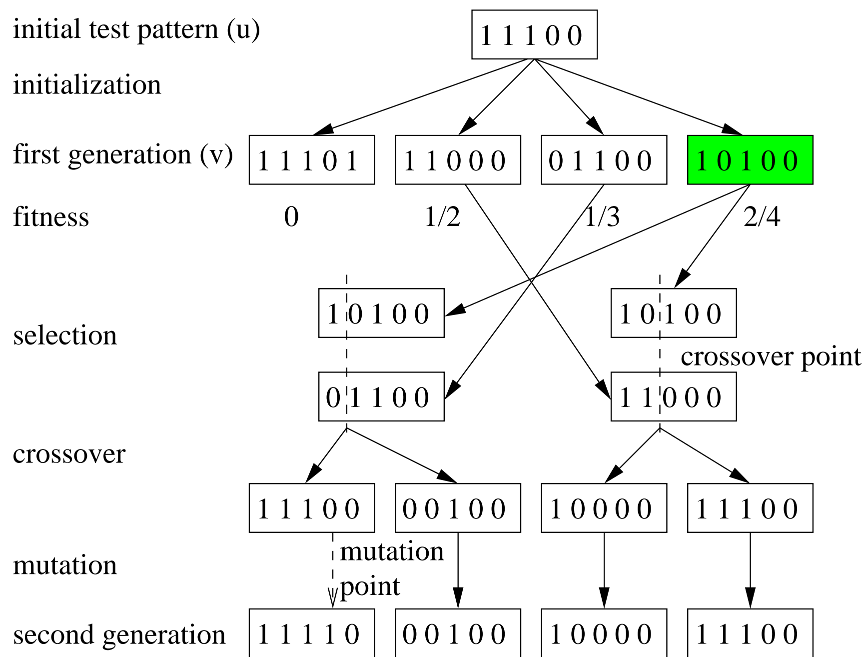


Figure 8-4. The first iteration of GA for generating the best second pattern for $u = 11100$.

8.2.2.1 Initialization

The first population is initialized with random test patterns that are similar to u_i. Each individual in the initial population has a Hamming distance k from u_i. During the experiments, we choose k to be $\max(0.004|u_i|, 1)$.

8.2.2.2 Fitness Computation

For each individual v, the golden design G is simulated with the pair of test patterns $<u_i,v>$. Then the fitness of v is computed by Equation 8-2. For example, the fitness values for four candidates are shown in Figure 8-4.

8.2.2.3 Selection

Selection is based on the fitness of each individual. An individual with a greater fitness is more likely to be selected. The selection shown in Figure 8-4 demonstrates that the individual with a greater fitness (such as 10100) are more likely to be selected than the one with a smaller fitness (such as 11101).

8.2.2.4 Crossover and Mutation

A single crossover point is randomly selected and crossover is performed on parents to produce two children. During mutation, a randomly selected position is mutated with a low mutation rate. For example, Figure 8-4 shows only 1 mutation for 4 individuals.

Although the Hamming distance to u_i is small in all individuals of the initial generation, crossover and mutation will increase the Hamming distance from generation to generation. Theoretically, the largest possible Hamming distance between the jth generation and u_i is at most $2^j * |k|$ considering only crossover. In order for all test patterns to be evaluated with some probability, the total number of generations should be large enough to allow $|u_i|$ Hamming distance. However, we may need only a small number of generations, as the affinity heuristic suggests. During experiments, we fix the number of generations to be 5. So, the maximum Hamming distance could be $2^5 \times 0.004|u_i|$ which is around $10\%|u_i|$. By exploring around u_i with a small Hamming distance, we expect to get high quality pairs efficiently. As demonstrated in Section 8.3.4 (Figure 8-8), the pairs of test patterns with small Hamming distances are effective in providing a significant improvement in sensitivity.

Since line 3-11 are performing an independent GA searching for each first pattern u_i, the body of the first for-loop can run in parallel similar to Algorithm 9. Therefore, C cores will give a speedup close to C. Combined with Algorithm 9 to generate first patterns, the whole process

can run in parallel. Given unlimited computing resources, we can use each core to generate one pair and combine all pairs together. This optimistic overall running time consists of compiling the design, generating one first pattern from Algorithm 9 and its corresponding second pattern from Algorithm 10.

8.2.3 Selection of $TriggerLimit$

As introduced in Section 8.1.3, there is a tradeoff between introducing switching in the rare nodes and minimizing switching in the golden design. We try to address this challenge by selecting a reasonable $TriggerLimit$. This section illustrates why it is a challenge to select a reasonable $TriggerLimit$ - a larger $TriggerLimit$ can lead to increased total switching (and reduced sensitivity), while a smaller $TriggerLimit$ can lead to reduced probability of Trojan activation.

$TriggerLimit$ in Algorithm 9 controls how many rare signals should be activated by each first pattern u. Its second pattern generated by Algorithm 10 tends to introduce as many switching in the rare nodes activated by u as possible to increase fitness value. First, we show that if the $TriggerLimit$ is too large, it may lead to sub-optimal patterns with lower sensitivity. Assume that the inserted Trojan has 8 trigger points and we compare $TriggerLimit = 128$ to $TriggerLimit = 8$. (i) With $TriggerLimit = 128$, suppose that we generate a first pattern u' that is able to activate 128 rare nodes (include all 8 trigger points of the Trojan), and find a second pattern v' by Algorithm 10 which introduces current switching in all of these 128 rare nodes with a total switching of 1000. (ii) With $TriggerLimit = 8$, suppose that we get a first pattern u^* that is able to activate only the 8 trigger points of the Trojan, and find a second pattern v^* which introduces current switching in all of these 8 rare nodes with a total switching of 500. When these two test vectors ($<u',v'>$ versus $<u^*,v^*>$) are applied to the Trojan inserted design, the switching from the Trojan will be same, but the sensitivity produced by $<u^*,v^*>$ will be greater than $<u',v'>$ since the original switching produced by $<u^*,v^*>$ is smaller. Next, we show that if $TriggerLimit$ is too small, it may be not beneficial for Trojan detection. In the above example, the probability of the 128 rare nodes

activated by u' containing all 8 trigger points of the Trojan is high, while the probability of the 8 rare nodes activated by u^* being the exact 8 rare triggers points of the Trojan is very low. It is obvious that if we introduce less switching in the Trojan area, the sensitivity will be smaller.

In summary, with a small $TriggerLimit$, e.g., 8 in the previous example, the probability of activating unknown Trojans is extremely low compared to $TriggerLimit = 128$ with the same number of test patterns. With a large $TriggerLimit$, e.g., 128 in the previous example, Algorithm 10 will try to maximize the fitness value by introducing as many switching in the rare nodes as possible, and as a result, introducing a large original switching and low sensitivity in Equation 8-1. In practice, we set $TriggerLimit$ to be a few times of the largest possible trigger points. $TriggerLimit$ is typically small since the number of trigger points is small, otherwise, it will introduce noticeable power or area overhead, which is easier for debug engineer to detect using side-channel analysis. In our experiments, $TriggerLimit$ is set to be 32 for Trojans with 8 trigger points.

8.3 Experiments

To evaluate the effectiveness of our approach, we did a variety of experiments to show the results on different benchmarks and compared the results to the state-of-the-art approach. In addition, we also evaluated the efficiency of our approach utilizing multi-core systems. Note that although we evaluate the effectiveness of the generated test patterns using gate-level simulation, the test patterns generated by our approach are also applicable on post-silicon designs (fabricated chips).

8.3.1 Experimental Setup

All algorithms of our framework are implemented in C++ and Algorithm 9 utilizes Z3 [121] C++ as our SMT solver. Since MERS [6] is the state-of-the-art (closest to our approach), we used the same benchmarks as MERS - a subset of ISCAS-85 [30] and ISCAS-89 [56] gate-level benchmark circuits. We have also used two large benchmarks, memory controller (MC) from TrustHub [105] and MIPS processor from OpenCores [96], to demonstrate the scalability of our approach. We performed a variety of experiments on a

machine with Intel Xeon E5-2698 CPU @2.20GHz. We compared three approaches with the configurations shown below:

1. **MaxSense**: MaxSense utilizes SMT (Algorithm 9) to generate the first patterns and genetic algorithm (Algorithm 10) to generate the second patterns. We fixed the number of test patterns $M = 5,000$ for ISCAS benchmarks, and $M = 10,000$ for MC and MIPS in Algorithm 9. In Algorithm 10, we set the number of individuals to be 200 in each generation, the number of generations to be 5, and mutation rate to be 0.1.
2. **NDT+GA** [5]: It utilized N-detect [12] to generate the first patterns and genetic algorithm (Algorithm 10, the same configuration as GA in MaxSense) to generate the second patterns. We fixed N=1000 for the N-detect criterion. The initial random test patterns in MERO consists of 100,000 random patterns for ISCAS benchmarks and one million random patterns for MC and MIPS. To increase the probability of activity in Trojan area, we flipped two rounds of each bit for every initial random test pattern.
3. **MERS-s** [6]: It is the state-of-the-art approach from [6] (MERS with simulation-based reordering) with the best settings (C = 5.0) [6]. We did not compare with random tests and MERS (with Hamming distance) since MERS-s outperforms them.

8.3.2 Generation of Hardware Trojans

To statistically evaluate the performance of different approaches, we first generated 1000 valid Trojans for each benchmark. To make these Trojans covert under traditional validation, we constructed each Trojan with a number of rare signals (trigger points) defined by a rareness threshold for each benchmark. To get the list of all rare signals, we first ran 1 million random simulations for each benchmark. We set rareness threshold to be 0.1 for ISCAS benchmarks and 0.005 for MC and MIPS over all experiments. The number of rare signals is shown in the second column of Table 8-1. As we can see, the number of rare signals are around 1000 for large benchmarks. After achieving the rare signal list, different combinations of rare signals are randomly sampled and fed into ATPG tool to check their validity. Finally, 1000 valid Trojans are independently inserted into the benchmarks to construct design under tests (DUTs). These constructed 1000 DUTs for each benchmark are used to evaluate the performance of different approaches over all the experiments. It is easy to see that the probability of activating these Trojans using random simulation is at most 10^{-p} for ISCAS benchmarks, and 200^{-p} for large benchmarks, assuming each Trojan is constructed by p independent rare signals. In all of our

experiments, the trigger points p is set to be 8. The possible combinations that an adversary can exploit to insert Trojans are in the order of $\binom{1000}{p}$ with 1000 rare signals, which is around 10^{19} for $p = 8$.

8.3.3 Performance Evaluation

We applied the test patterns generated by the three approaches in Section 8.3.1 to simulate both the golden design and all DUTs in Section 8.3.2. We computed the side-channel sensitivity in current switching according to Equation 8-1. For each DUT, we conclude the existence of a Trojan if the sensitivity is greater than 10% [81].

The results are shown in Table 8-1. For each approach, we report the sensitivity, the percentage of detected Trojans and the overall running time. The sensitivity is the average sensitivity over 1000 randomly sampled Trojans, each of which is computed using Equation 8-1 with all the test patterns. The percentage of detected Trojans shows the fraction of Trojans whose sensitivities are above the threshold 10%. The running time of MERS-s consists of the generation of MERS test patterns and simulation-based reordering. The running time of NDT+GA and MaxSense consist of the running time of N-detect [12] and SMT (Algorithm 9), respectively, and the running time of genetic algorithm (Algorithm 10). The entry marked with dash represents test generation timeout after one week.

Table 8-1. Comparison of MaxSense with NDT+GA [5] and MERS-s [6] over 1000 Trojans.

		MERS-s [6]			NDT + GA [5]				MaxSense				
Bench	#rare signals	Sens-itivity	% of Trojans detected	Time (h)	Sens-itivity	Imp. over [6]	% of Trojans detected	Time (h)	Sens-itivity	Imp. over [6]	% of Trojans detected	Time (h)	Imp. over [6]
c2670	43	5.6%	2.4%	5.1	110%	19.6x	100%	1.1	133%	23.8x	100%	0.3	4.6x
c5315	164	1.5%	0.3%	23.5	52.4%	34.9x	98.3%	4.1	163%	108.7x	99.4%	0.9	26.1x
c7552	278	1.7%	0.2%	40.3	34.5%	20.3x	82%	7.9	51.9%	30.5x	83.5%	1.4	5.1x
s13207	604	1.7%	0%	11.5	79.7%	46.9x	100%	12.9	99%	58.2x	100%	0.7	16.4x
s15850	649	1.0%	0%	16.1	54.3%	54.3x	99.5%	15.1	55.7%	55.7x	97.1%	0.9	17.9x
s35932	1152	1.0%	0.1%	10.7	52.9%	52.9x	66.6%	29.6	95.7%	95.7x	99.8%	1.8	5.9x
MC	1306	-	-	-	-	-	-	-	14.9%	∞	85.2%	5.7	∞
MIPS	906	-	-	-	-	-	-	-	35.1%	∞	55.2%	13.0	∞
Avg. [1]	-	2%	0.5%	17.9	64.0%	38.2x	91.1%	11.8	99.7%	62.1x	96.6%	1	12.7x

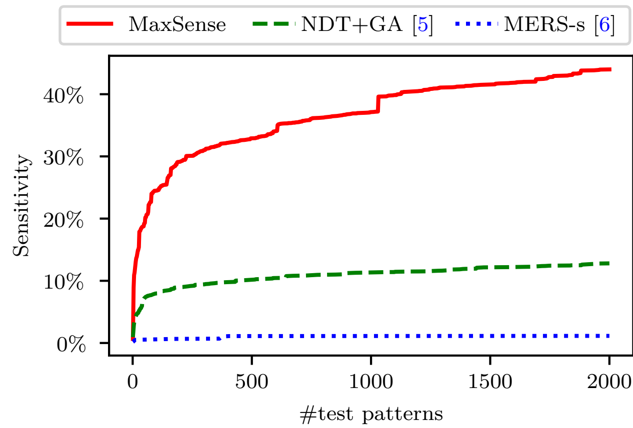
[1] Since MERS-s [6] and NDT+GA [5] cannot finish in the MC and MIPS, all average results are computed over ISCAS benchmarks for comparison.

8.3.3.1 Sensitivity comparison

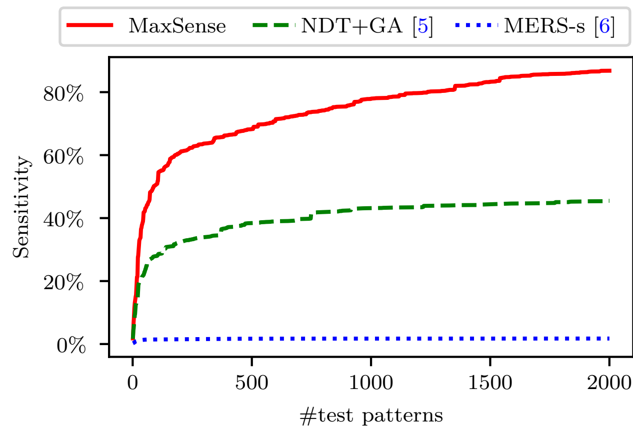
The sensitivity is the most important indicator of the quality of the generated test patterns, since a larger sensitivity can help detect more Trojans directly or by sophisticated classification approaches. As shown in Table 8-1, the overall sensitivity of MaxSense outperforms MERS-s by 62 times, and the NDT+GA improves a factor of 38 over MERS-s. For the ISCAS benchmarks, while the test patterns generated by MERS-s only get less than 2% sensitivity except for the smallest one c2670, NDT+GA and MaxSense improve the sensitivity to 64% and 99.7% on average, respectively. For the two large benchmarks, MERS-s and NDT+GA cannot finish within one week time limit due to the long running time of N-detect approach in these two approaches and also reordering in MERS-s. On the other hand, MaxSense is able to provide 14.9% and 35.1% sensitivity in MC and MIPS, respectively, with a few hours of test generation time.

By comparing the performance of NDT+GA and MaxSense, we can observe that SMT-based approach achieves 63% higher sensitivity compared to N-detect based approach. It shows that the first patterns generated by SMT is better than N-detect in activating rare nodes. While N-detect approach increases the activation of rare nodes by flipping bits from random test patterns, SMT-based approach directly controls which parts of rare nodes to be activated. Therefore, it is expected that SMT-based approach would outperform N-detect based approach. Compared to reordering technique in MERS-s, our genetic algorithm generates the most profitable second patterns for a given first pattern, leading to a significant reduction in the original switching, which will be explained in Section 8.3.4.

To inspect the effectiveness of one test pattern pair, we compute the average sensitivity of all DUTs from c7552 and s13207 after applying the first 2,000 test pattern pairs in Figure 8-5. Both Figure 8-5A and Figure 8-5B reveal the same pattern that the average sensitivity grows significantly faster by the test patterns from MaxSense than NDT+GA and MERS-s. When the number of test patterns is below 500, MaxSense is already able to generate more than 30% and 60% average sensitivity in c7552 and s13207, respectively, while NDT+GA achieves 10%



A c7552



B s13207

Figure 8-5. The average sensitivity of two benchmarks with respect to the length of tests.

and 40% in these two benchmarks. The average sensitivity of NDT+GA grows fast in the first few hundreds of test patterns, but it saturates soon. MERS-s performs the worst, with less than 2% in both benchmarks after 2,000 test patterns. Therefore, the test patterns generated by MaxSense is more effective and more compact than NDT+GA and MERS-s. Among these three approaches, the test patterns generated by MERS-s have the worst quality that achieve an average sensitivity typically less than 2%, which is far less than process variation and environment noise.

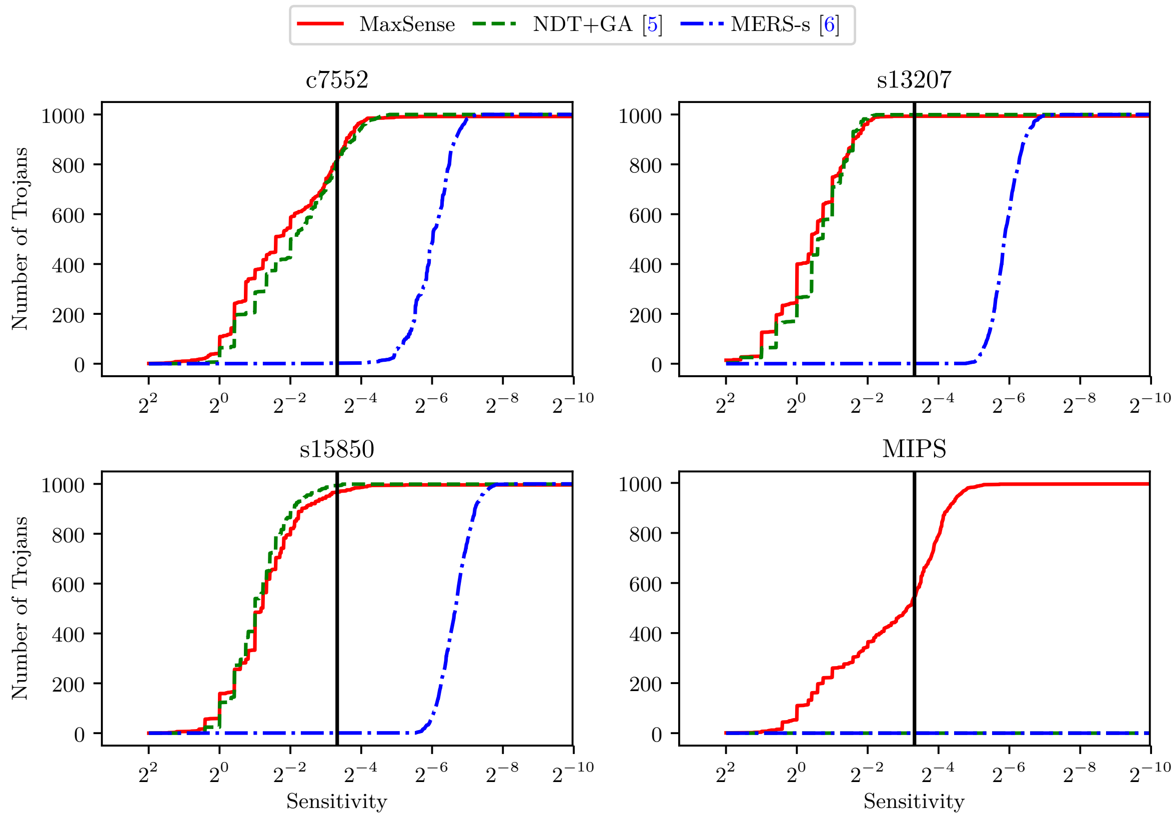


Figure 8-6. The distributions of sensitivities by three approaches over 1000 Trojans.

8.3.3.2 Detected Trojans

With the assumption of 10% sensitivity threshold [81], the percentage of detected Trojans are shown in Table 8-1. Since the sensitivity from MERS-s are mostly less than 2%, MERS-s missed almost all the Trojans. On the other hand, NDT+GA and MaxSense are able to detect 91.1% and 96.6%, respectively, of Trojans on ISCAS benchmarks.

The cumulative distributions of the sensitivities over 1000 Trojans in c7552, s13207, s15850 and MIPS are shown in Figure 8-6. The x-axis represents the sensitivity, y-axis represents the number of Trojans that have sensitivities greater than x, and the vertical line represents 10% sensitivity. For example, in s13207, almost all the Trojans have sensitivities greater than the sensitivity threshold in both MaxSense and NDT+GA, while in MERS-s this number is 0. In other words, if we assume the process variation to be 10%, MaxSense and

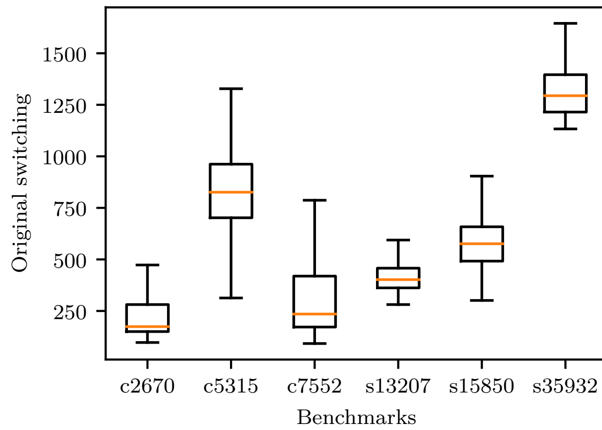
NDT+GA can detect the majority of these randomly sampled Trojans with high confidence, while MERS-s missed almost all of them. The exact numbers are reported in Table 8-1. Overall, our approach can detect majority (over 90% on average) of the Trojans in most of the benchmarks due to the higher sensitivities provided by our test patterns, whereas the test patterns generated by MERS-s can only detect 0.5% in ISCAS benchmarks on average.

8.3.3.3 Test generation time

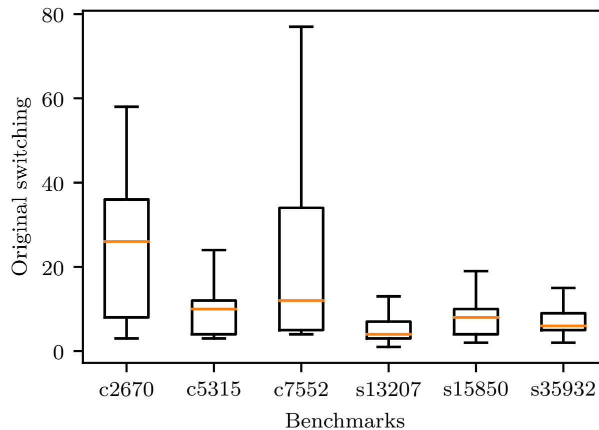
Finally, we compare the test generation time of the three approaches. As shown in Table 8-1, while MERS-s and NDT+GA requires 17.9 and 11.8 hours on average, respectively, to generate test patterns for ISCAS benchmarks, MaxSense takes only 1 hour. MaxSense is up to 26.1x, 12.7x on average, more efficient than MERS-s. For the two large benchmarks, MC and MIPS, MERS-s cannot finish in one week, while MaxSense can generate high quality test in several hours. It indicates that the improvement increases when the size of design becomes larger and larger. Furthermore, MaxSense is able to utilize multi-core platforms to reduce the running time by several times for large designs (see Section 8.3.5), while MERS-s is not suitable for even medium-size designs. The long running time of MERS-s is due to the usage of N-detect approach in generating test patterns, and time-consuming simulations in reordering test patterns. The reordering in MERS-s takes $O(n^2)$ simulations for each pair of test patterns, where n is the number of test patterns. On the other hand, the number of simulations in our genetic algorithm is $O(n)$, where n is the number of first test patterns, since we evaluate 5 generations and 200 individuals for each generation. Moreover, the test patterns generated by MaxSense is more compact than MERS-s as shown in Figure 8-5. Therefore, the linear growth in the number of simulations in Algorithm 10 leads to significantly faster test generation than the quadratic growth in the number of simulations in the reordering part of MERS-s.

8.3.4 Evaluation of Original Switching

Because the inserted Trojans are usually tiny compared to the whole design and the probability of fully activating a Trojan is low, it is critical to minimize the current switching in the original design such that the sensitivity is high enough to be detected. Figure 8-7 shows



A MERS-s [6]



B MaxSense

Figure 8-7. The distribution of the original switching in the golden design.

the box plot of all original switching from test patterns generated by MERS-s and MaxSense for ISCAS benchmarks. Figure 8-7A shows that the original switching from MERS-s is in the order of several hundreds, up to 1500. Figure 8-7B shows that MaxSense achieves less than 100 original switching for all benchmarks, with the average original switching around 10 to 20. Compared to MERS-s, MaxSense is able to achieve up to more than 100 times reduction in the original switching, e.g., the median of original switching in s35932 is around 1250 from MERS-s and less than 10 from MaxSense. With an 8-trigger Trojan, the number of extra current switching is typicall less than 16, assuming the Trojan is not fully activated. Therefore,

the large original switching becomes the main bottleneck that prevents MERS-s from achieving a high sensitivity.

The large original switching from MERS-s is due to its reordering technique. As the reordering of MERS-s restricts each test pattern to find its pair from the generated test patterns, the minimum original switching that reordering can achieve is bounded by the optimum pairs inside these test patterns. On the other hand, MaxSense fixes the first pattern and searches an open space for profitable second patterns to minimize the original switching. The searching process starts with test patterns that are close to the first pattern, and gradually increases the distance using genetic algorithm. For each first test pattern u from SMT (Algorithm 9), the best second pattern v is found by GA with 5 generations. Thus, the maximum possible Hamming distance between u and v can be as large as $10\%|u|$ (see Section 8.2.2). It follows the motivation of affinity heuristic introduced in Section 8.1. We examined the hamming distance between the generated pairs of MaxSense in Figure 8-8. It is interesting to see that almost all of the distances are 1 in the combinational circuits from ISCAS-85 [30]. They reveal a similar property as the example in Figure 8-2 that one bit change can maximize the activity in the Trojan area and minimize the activity in the remaining of the design. For the ISCAS-89 [56] benchmarks, most of the distances are less than 3 except for s35932. The results hint that a small number of generations (faster runtime) in genetic algorithm can provide significant sensitivity improvement.

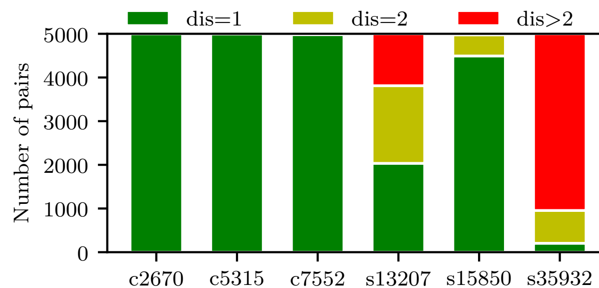


Figure 8-8. Hamming distance of all pairs of test patterns by MaxSense.

8.3.5 Concurrency of MaxSense

As discussed in Section 8.2, N-detect approach cannot run in parallel since the N-detect criterion relies on the overall performance of all test vectors. Although the second step of NDT+GA can run in parallel, it does not benefit much from multi-core platforms since N-detect part consumes the majority of test generation time. Similarly, MERS-s cannot utilize multi-core platforms because of the N-detect criterion. What is worse, the reordering step of MERS-s cannot run concurrently either. As a result, only MaxSense is evaluated in this experiment, whose both steps can run in parallel.

To evaluate the performance of MaxSense in a multi-core platform, we tested the test generation time of MIPS with 1, 2, 4, 8, and 16 threads running in different cores. Multi-threading scheme is implemented using C++ pthread library. Each thread is responsible to generate its own pairs of test patterns. For example, to generate 10,000 pairs of test patterns in a 16-core platform, each thread is responsible to generate 625 pairs of test patterns using Algorithm 9 with $M = 625$ followed by Algorithm 10. The overhead of multi-threading includes compiling the design for individual simulation of each thread and returning the generated test pattern pairs to the main thread.

Figure 8-9. The test generation time of MaxSense with multi-core platforms for MIPS processor.

The test generation time of MaxSense is shown in Figure 8-9. It is clear to see that MaxSense can achieve better performance in multi-core platforms since both its pairs of test patterns can be generated in parallel. With 2 threads, MaxSense can achieve speedup around

1.6x. The final speedup using 16 cores is more than 6 times, leading to less than 2 hours in generating 10,000 test pattern pairs. It is significantly faster than MERS-s which takes longer than one week to finish. With enough cores, the best possible performance would be the total time for compiling the design and generating one pair of test patterns. Since compilation time is the dominant factor and it is linearly related to the design size, the time complexity of MaxSense would be linear with respect to the design size with enough cores.

8.4 Summary

Side-channel analysis provides a promising approach for Trojan detection. The state-of-the-art test generation technique (e.g., MERS-s [6]) is not beneficial for large designs due to its high runtime complexity. Most importantly, the sensitivity obtained by the existing approaches is very low compared to environmental noise and process variations, making them useless in practice. Our proposed approach addresses both limitations by developing an SMT-based first pattern generation algorithm and a genetic algorithm based second pattern generation algorithm that can increase the sensitivity drastically while significantly reduce the test generation time. Our approach breaks down the problem into two sub-problems. The first task generates effective test patterns to maximize the excitation of rare values. The second task finds the best matching pair for each test pattern generated in the first task to maximize the sensitivity. In this chapter, we demonstrated that the combination of the SMT-based approach with the genetic algorithm can generate significantly better test patterns than MERS-s. Our proposed test generation approach can improve both side-channel sensitivity (up to 109x, 62x on average) and test generation time (up 26x, 13x on average) compared to MERS-s. Experimental results demonstrated that our approach can detect the majority of Trojans in the presence of process variation and noise margins while the state-of-the-art approaches fail.

CHAPTER 9
TROJAN DETECTION USING DELAY-BASED SIDE-CHANNEL ANALYSIS

Side-channel analysis is widely used for hardware Trojan detection in integrated circuits by analyzing various side-channel signatures, such as timing, power and path delay. Compared to current-based side-channel analysis in Chapter 8, delay-based side-channel analysis is beneficial as the delay of each output can be measured independently, and an inserted Trojan may affect multiple observable outputs. Existing delay-based side-channel analysis techniques have two major bottlenecks: (i) they are not suitable in detecting Trojans since the delay difference between the golden design and a Trojan inserted design is negligible, and (ii) they are not effective in creating robust delay signatures due to reliance on random and ATPG based test patterns.

In this chapter, we propose an automated approach to generate high quality test patterns for path delay based side-channel analysis to significantly improve the side-channel sensitivity. The main observation is that the tests generated by logic testing are more likely to activate trigger conditions, and by utilizing these tests, we can produce two completely different critical paths for the same register in the golden design and in a Trojan-inserted design. As a result, it can lead to significantly different path delays, compared to the negligible delay introduced by few extra gates (from a Trojan) in a fixed critical path. In this chapter, we make the following three important contributions:

1. We propose an efficient test generation method to maximize observable path delays by changing critical paths.
2. We design a lightweight and effective logic testing algorithm to generate tests for delay-based side-channel analysis. The generated tests assume no preliminary information about critical paths or trigger conditions.
3. We perform a fast and efficient Hamming-distance based reordering of the generated tests to maximize the delay deviation between the golden design and Trojan inserted design. We design a distance evaluation method to increase the probability of constructing a critical path from the trigger to the payload.

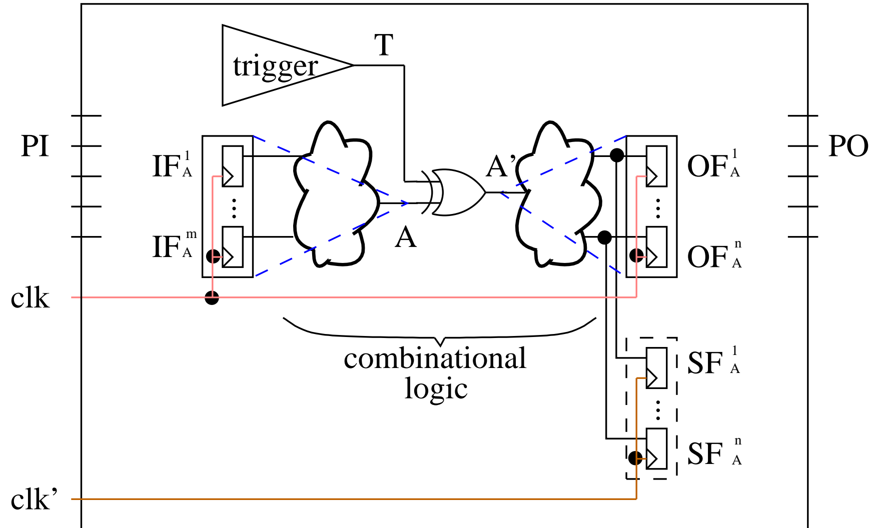


Figure 9-1. Path delay measurement using shadow registers [15].

The remainder of the chapter is organized as follows. Section 9.1 describes our automated test generation approach. Section 9.2 presents the experimental results. Finally, Section 9.3 concludes the chapter.

9.1 Test Generation for Path Delay Analysis

The main challenge in Trojan detection using delay-based side-channel analysis is how to increase the observability. One of the common methods to measure path delay is using shadow registers [15]. As shown in Figure 9-1, the original registers and shadow registers utilize different clocks to measure delays by controlling the skew of *clk* and *clk'*. The original *clk* is used to maintain the correct functionality, while the second *clk'* can be tuned to find out the exact time of a signal flipping by comparing the values in corresponding registers. As a result, there would be no delay if the signal value does not change between two simulations. For example, when the value of OF_A^1 remains the same between two simulations, SF_A^1 will have the same value as OF_A^1 irrespective of how clk' is tuned, thus, no delay information can be retrieved.

To observe the delay caused by the inserted Trojan, the critical path of some register in the output layer[1] of A, e.g., OF^1_A in Figure 9-1, needs to contain A'. Otherwise, the delay between the input layer[2] and the output layer will be almost the same between the golden design and Trojan-inserted design (only differed slightly due to capacitance change). With the critical path crossing A', the signal value of A' has to switch to reveal delay information, either by trigger T or by asset A. In addition, there must exist a path from A' to the output layer where all signals need to switch. Our goal is to generate test vectors that are able to maximize the delay difference of a critical path from the Trojan to the output layer.

9.1.1 Test Generation for Path Delay Maximization

The activation of a trigger is important in maximizing the delay difference. Existing approaches try to find critical paths that are affected by the Trojan. However, without the activation of a trigger, the delay difference is at most one gate difference. As shown in Figure 9-2, the trigger signal T remains zero and the Trojan-inserted design behaves exactly the same as the original design. As a result, any delay information from the input layer to T is hidden and the delay of A' is determined by A. Assume that we are able to construct a critical path from A to the output layer using a specific test vector. Since the behaviors of the golden design and the Trojan-inserted design are the same, two critical paths are the same except for the extra XOR gate. On the other hand, the critical paths can change significantly when the trigger is activated. Figure 9-3 shows the optimal scenario of maximizing the delay difference. In Figure 9-3, the critical path in the Trojan-inserted design goes through the trigger T and propagate the delay to the output layer, which is completely different from the path in the golden design. As a result of two totally different critical paths, the measured delay difference

[1] The **output layer** of a signal contains all the registers encountered in the immediately succeeding layer in the path from the signal to primary outputs.

[2] The **input layer** of a signal contains all the registers encountered in the immediately preceding layer in the path from the signal to the primary inputs.

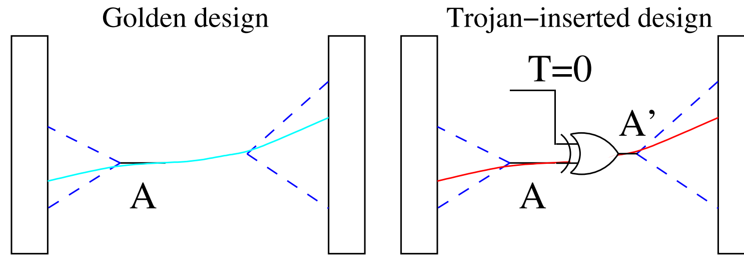


Figure 9-2. The small delay difference by existing approaches with the same critical path.

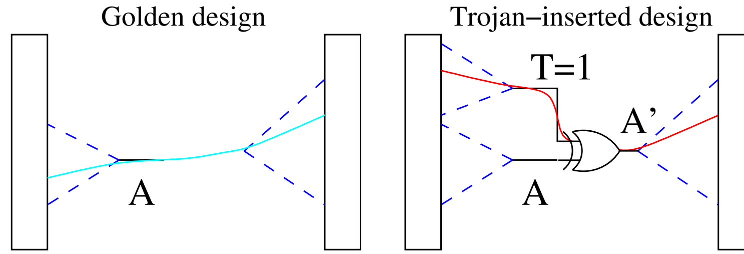


Figure 9-3. Our approach maximizes delay difference by changing critical paths.

in the output layer can be significantly larger, compared to the scenario when the trigger is not activated in Figure 9-2.

Therefore, the goal of our test generation technique is to increase the probability of activating trigger conditions. As the attackers are more likely to construct trigger conditions using rare signals, we propose to use a SAT-based approach to generate test patterns in Algorithm 11. It first parses the circuit and computes logic expressions for all rare signals. Then, it repeats k times to generate k test vectors, where k is defined by the user to balance debug time and performance. In the ith iteration, we first randomize the order of rare nodes such that each time the generated tests can cover different sets of rare nodes. Next, we keep adding rare nodes into current trigger CT if CT is still valid. Finally, we use a SAT solver to return a test for CT. Intuitively, we want to generate a test that is able to activate as many rare nodes as possible. Since an adversary wants to hide from side-channel analysis, i.e., introduce the minimum delay, the number of trigger points is typically small. The test that is able to activate many rare nodes has the high probability of covering an unknown trigger condition. Note that the goal of our test generation partially overlaps with logic

testing, without the requirement of propagating the effects of payload to the primary output. Experiments show that our lightweight algorithm is effective in delay-based side-channel analysis. Our framework is expected to perform better in the presence of advanced logic testing techniques.

Algorithm 11 Test Generation

```
1: procedure TestGeneration(circuit netlist, a set of rare nodes (R), the number of test
   vectors k)
2:     Parse circuit netlist, and compute logic expression for each rare node
3:     Initialize T = {}
4:     i = 1
5:     while i <= k do
6:         Current trigger CT = ∅
7:         Randomize the order of rare nodes R
8:         for rare node r ∈ R do
9:             if CT ∪ r is a valid trigger then
10:                CT = CT ∪ r
11:            end if
12:        end for
13:        Solve CT and get a test t
14:        t_i = t
15:        i = i + 1
16:    end while
17:    return test vectors T = {t_1, t_2, ..., t_k}
18: end procedure
```

9.1.2 Hamming-distance based Reordering

Activating the trigger is not a sufficient condition to introduce delay of the Trojan to the output layer. It also requires construction of a critical path from the Trojan to the output layer. This is a strict condition due to the following reasons. First, the trigger signal T has to switch between two consecutive simulations. Otherwise, the critical path will not pass through the trigger signal. Next, every signal in the critical path has to switch. Figure 9-4 shows an example to illustrate the difficulty of creating a critical path from the trigger T to the output layer. Assume that the payload A' flips from 0 to 1 due to the activation of the trigger. In order to propagate the delay, the signal P has to flip from 0 to 1, which requires signal N to have value 0 in the beginning. When we consider all the signals in a path from A' to the

output layer, more and more constraints need to be applied. Directed test generation, such as ATPG or SAT-based approach, can be used to find the optimal solution when the payload is known. However, as we do not know the exact place of the trigger and payload a priori, these approaches may not work. In this section, we propose a probabilistic approach to increase the likelihood of constructing such a critical path using Hamming-distance based reordering.

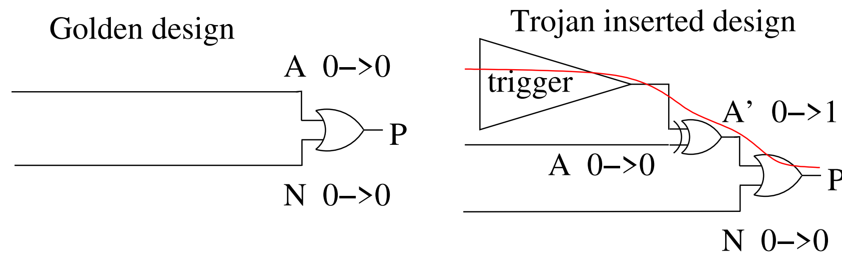


Figure 9-4. The constraints to ensure a critical path from the trigger to the output layer.

Algorithm 12 shows our reordering approach to statistically create a critical path and maximize sensitivity. The main idea is to find a test vector that differs from the current test vector mostly as its successor. We define the *distance* of two vectors as the summation of two parts. The first part is the Hamming distance of the *feature vector*, which represents the activation status of all rare signals. For example, assuming a test t is able to activate the first three rare signals out of four rare signals in a design, then its feature vector is 1110. With a larger difference in the feature vector of two test vectors, one trigger condition is less likely to be activated by the two vectors simultaneously. The second part is the Hamming distance of the test vectors. Large Hamming distance between the test vectors increases the probability of signal switches in the cone area impacted by A'. As a large difference in the features vectors of two tests t_i and t_j typically implies a large Hamming distance of these two test vectors, we add a small weight (0.1 in Algorithm 12) to the Hamming distance of test vectors (the latter part). As shown in Algorithm 12, we first simulate the design and compute the feature vector for all test vectors. For each test vector t_i, we try to find the test vector with the largest distance among the remaining ones as its successor (line 8-17). After finding the test vector, we swap it with t_{i+1} (line 18).

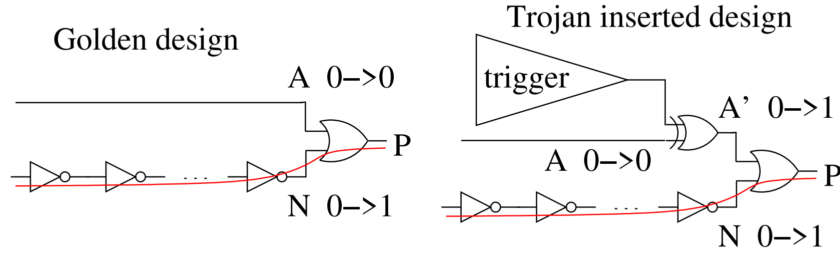


Figure 9-5. A longer path may mask the delay from the Trojan.

The Hamming-distance based reordering is efficient, with k simulations and $O(k^2)$ computations of Hamming distance, where k is the number of generated test patterns. As the Trojan is unknown, the generated tests may not be able to sensitize the critical path from A' to the output layer. For example, when the signal N in the longer path switches, the delay of P is determined by N, which masks the delay from the Trojan as shown in Figure 9-5. As a result, there would be no difference between the delays from golden design and Trojan inserted design. In general, for some path from A' to the output layer, all neighbor signals with longer delays need to remain the same value. However, without knowing the exact Trojan, this requirement is hard to fulfill. Fortunately, as an attacker is likely to construct a hard-to-activate trigger condition, the path from the input layer to the trigger T is typically long. It potentially produces large delay in the trigger signal T, which leads to detection of Trojans as demonstrated in Section 9.2.

9.2 Experimental Results

9.2.1 Experimental Setup

Implementation: All of our algorithms and simulators are implemented in C++. The SAT expressions in Algorithm 11 are solved using Z3 [121]. The experiments are conducted using a machine with Intel Xeon CPU E5-1620 v3 @ 3.50GHz and 16GB RAM.

Benchmarks: To evaluate the effectiveness of our approach in detecting hardware Trojans, we selected five sequential benchmarks from ISCAS-89 [56], as well as a large benchmark MIPS from OpenCores [96]. Trigger conditions are constructed using rare signals. For the two small ISCAS-89 benchmarks, s1196 and s1423, each trigger condition

Algorithm 12 Hamming-distance based Reordering

```
1: procedure REORDER(circuit netlist, test vectors T = {t_1, t_2, ..., t_k})
2:     for t_i in T do
3:         Simulate the netlist with t_i
4:         Set feature vector of t_i: each bit of fv_i represents whether a certain rare signal is
   activated or not
5:     end for
6:     Set weight ω = 0.1
7:     for i = 1 to k do
8:         Initialize best successor for t_i as bestSuccessor = −1
9:         Initialize the largest distance as maxdist = −1
10:        for j = i + 1 to k do
11:            The distance of feature vector dist1 = Hamming(fv_i, fv_j)
12:            The distance of test vectors dist2 = Hamming(t_i, t_j)
13:            if dist1 + ω * dist2 > maxdist then
14:                maxdist = dist1 + ω * dist2
15:                bestSuccessor = j
16:            end if
17:        end for
18:        Swap the test vectors of t_{i+1} and t_{bestSuccessor}
19:    end for
20:    return reordered test vectors T
21: end procedure
```

is constructed by 4 trigger points, while Trojans of the other benchmarks are constructed by 8 trigger points. All trigger points are selected from rare nodes from the design, where the rareness thresholds are 0.1 for ISCAS benchmarks and 0.005 for MIPS. The total number of rare nodes is listed in Table 9-2. For each benchmarks, 1000 Trojans are randomly sampled. Each Trojan is inserted into the golden design to form one DUT. In other words, there are 1000 DUTs for each benchmark to evaluate the performance.

9.2.2 Path Delay Computation

The path delay can be measured using static timing analysis of gate-level models. We first compiled the benchmarks using Synopsys Design Compiler. Next, we generated Standard Delay Format (SDF) file that contains delay information of each gate and net in the design by linking with saed 90nm library [124]. Finally, SDF files are back-annotated into our simulator. The simulator simulates all DUTs with generated test patterns, and reports delay information

computed using corresponding SDF files. Due to many factors in manufacturing steps, there are process variations in ICs, resulting in different delay fingerprints of the same design. To reflect the process variations, we added $\pm 7.5\%$ random variations to the SDF file of each DUT [72].

9.2.3 Evaluation Criteria

To evaluate the effectiveness of the generated tests by all approaches in detecting Trojans, we first simulated the golden design with the tests, and got the delay information of all registers. We use $dl^{f}_{gold}(t)$ to denote the delay for the register f of the golden design when simulating test pattern t. Then, we simulated each DUT with these tests, and got the delay information of all registers. Similarly, we use $dl^{f}_{dut}(t)$ to denote the delay for the register f in the DUT when simulating test pattern t. Finally, the maximum difference between the two delays which belong to the same register f is reported as our metrics to evaluate the performance of the tests from all approaches in (9-1).

$$diff = \max_{t,f}(|dl^{f}_{dut}(t) - dl^{f}_{gold}(t)|) \tag{9-1}$$

Assume that the test vector t^* produces the maximum delay difference in the register f^* for a given DUT, i.e., achieves the largest metric in (9-1). We define the following symbols for the ease of illustration:

- **OrigDelay (OD):** the delay of f^* in the goden design when applying t^*, i.e., $dl^{f^*}_{gold}(t^*)$.
- **Sensitivity:** the relative difference of delays in golden design and DUT, i.e., $diff/dl^{f^*}_{gold}(t^*)$

9.2.4 Statistical Evaluation

Table 9-1 summarizes experimental results from the application of our approach on the benchmarks compared to random test vectors and ATPG test vectors. For random simulation, we generated 10K random vectors for each benchmark. The number of random test vectors is selected to balance the overall performance and simulation time. To generate ATPG test vectors for path delays, we utilized TetraMAX with all delay faults and full sequential mode. For our approach, we fix the number of test vectors to be 1000 for all benchmarks,

i.e., $k = 1000$ for Algorithm 11. For each approach, Table 9-1 summarizes the number of test vectors (#), OrigDelay, delay difference ($diff$), and the average sensitivity over 1000 randomly sampled Trojans. From the results, we can see that random test vectors and ATPG achieve high delay sensitivity in small designs. However, the sensitivity produced by these two approaches are within 5% for two large benchmarks s38417 and MIPS, which is typically introduced by the noise. In contrast, our approach is able to achieve high sensitivity consistently. Overall, our approach can achieve 16 and 18 times improvement of sensitivity in delay based side-channel analysis over random test vectors and ATPG, respectively.

With the huge improvements in delay difference, our approach is able to detect more Trojans. In this experiment, a simple approach is used to declare the existence of a Trojan: if the delay in a DUT deviates from the delay in the golden design by more than the noise threshold (7.5%), then we declare that a Trojan exists in the DUT. Figure 9-6 shows the number of detected Trojans by these approaches. Among the 1000 randomly sampled Trojans, random simulation and ATPG are able to detect reasonable number of Trojans in the small design. However, the performance of these two approaches is poor for large designs, which detect less than 3% of all Trojans. On the other hand, our approach is able to detect more than half of the all Trojans for all benchmarks.

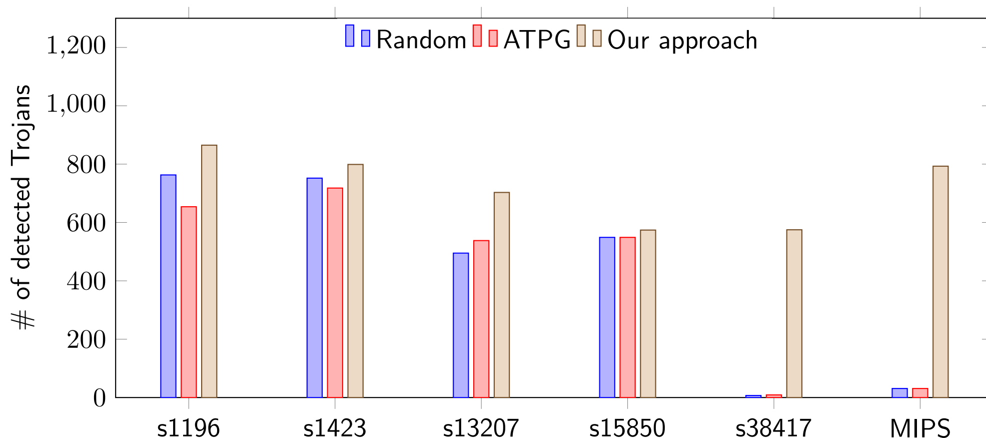


Figure 9-6. The number of detected Trojan given the noise of $\pm 7.5\%$ noise.

Table 9-1. Performance comparison of our approach with random simulation and ATPG over 1000 randomly sampled Trojans.

bench	Random #	OD (ps)	*diff* (ps)	sensi-tivity	ATPG #	OD (ps)	*diff* (ps)	sensi-tivity	Our Approach #	OD (ps)	*diff* (ps)	sensi-tivity	impro. /Random	impro. /ATPG
s1196	10K	1347	702	52%	221	1622	415	26%	1000	1073	1221	114%	2.2x	4.4x
s1423	10K	1586	313	20%	103	1385	173	12%	1000	675	1456	216%	11x	17x
s13207	10K	2108	169	8%	411	1553	144	9.3%	1000	1478	931	63%	7.9x	6.8x
s15850	10K	2370	192	8.1%	472	2149	178	8.3%	1000	2249	682	30%	3.7x	3.7x
s38417	10K	31826	1279	4%	1169	28729	1161	4%	1000	14768	11738	80%	20x	20x
MIPS	10K	62998	2495	4%	1363	61751	2446	4%	1000	21156	18227	86%	22x	22x
average	10K	17039	858	5%	623	16198	753	4.6%	1000	6900	5709	83%	16x	18x

Note that the performance of random simulation and ATPG is becoming worse when the design becomes large. It is due to the fact that the path between the input layer and the output layer in the small designs are relatively small, typically consisting of less than 10 gates. Therefore, an extra XOR gate from the Trojan can introduce reasonable delay difference to the delay of the output layer compared to 7.5% noise. However, with the number of gates increases in the paths, the effect of an extra gate becomes negligible. In contrast, our approach achieves consistent good performance in the all designs, due to the selection of test vectors that are likely to change the critical paths for the output layer entirely, as shown in Figure 9-3. In large designs, the change of critical paths is more likely to introduce drastically different delays.

Table 9-2. Test generation time of our approach in all benchmarks.

bench	#gates	#wires	#rare	Algo. 1	Algo. 2	total
s1196	550	568	195	33.6s	0.2s	33.8s
s1423	456	502	50	26.5s	0.05s	26.6s
s13207	2335	2504	604	150.8s	0.5s	151s
s15850	2812	3004	649	352s	0.5s	353s
s38417	23815	23844	3103	6195s	2.4s	6197s
MIPS	18123	18343	906	1058s	1s	1059s
Average	8015	8128	918	1303s	0.8s	1304s

The running time of our approach is shown in Table 9-2. The results show that our approach is efficient in generating test vectors for both ISCAS benchmarks and MIPS. For all benchmarks except for s38417, the total test generation time is within 20 minutes. This relatively longer time for s38417, which is less than 2 hours, is because that the number of rare nodes in s38417 is more than three times the number of rare nodes in all the other benchmarks. Overall, our approach can generate 1000 test vectors efficiently.

One major problem of gate-level simulation is the slow simulation speed. Therefore, the compactness of generated tests is critical to reduce the overall debug time. When the generated test patterns are not compact, it usually consumes a lot more time in simulation than in generating tests. From Table 9-1, 1000 test vectors generated by our approach are significantly better than 10K random vectors in both coverage and compactness of tests. While

the tests generated by ATPG are slightly more compact in small benchmarks, its performance is the worst among the three approaches.

9.3 Summary

Hardware Trojans are threats to assets in integrated circuits. To detect hardware Trojans, side-channel analysis is a widely used approach. Existing path delay based side-channel analysis techniques are not effective since the difference in path delays between the golden design and Trojan-inserted design is negligible compared to process variation and environmental noise margins. We presented an automated test generation approach to take advantage of logic testing in maximizing the difference in path delays. Compared to existing research efforts that fixes one critical path, our approach explores two different critical paths for the same register in the two designs, resulting in significantly large difference in path delay. Our experimental results using a diverse set of benchmarks demonstrated that our approach outperforms state-of-the-art path delay based side-channel analysis techniques. Specifically, our approach is able to detect most of the Trojans while state-of-the-art techniques fail to detect most of them in large designs when process variation and noise margin is higher than 7.5%.

CHAPTER 10
CONCLUSIONS AND FUTURE WORK

System-on-Chip (SoC) is the brain behind the computing devices today. Unlike microcontroller based designs in the past, even resource constrained Internet-of-Things (IoT) devices nowadays incorporate one or more complex SoCs. Drastic increase in SoC complexity has led to significant increase in SoC design and validation complexity. Reusable hardware IP based SoC design has emerged as a pervasive design practice in the industry to dramatically reduce design and verification cost while meeting aggressive time-to-market constraints. Growing reliance on these pre-verified hardware IPs, often gathered from untrusted third-party vendors, severely affects the security and trustworthiness of SoC computing platforms. Hardware-level vulnerabilities should be fixed before deployment since it affects the overall system security. This dissertation described a set of novel test generation approaches for SoC security validation covering both simulation-based validation and side-channel analysis.

10.1 Conclusions

In Chapter 3, I defined seven classes of SoC security vulnerabilities. Based on these vulnerabilities, I proposed a framework for generating security assertions. Using a diverse set of benchmarks, I demonstrated that the functional assertions generated by state-of-the-art assertion generation technique cannot eliminate the need for the dedicated security assertions. Specifically, these security assertions are able to detect all the implanted security vulnerabilities while the state-of-the-art method failed to detect most of them. I envision that the SoC designers will embed security assertions in their designs in the near future as part of their assertion-based security validation methodology. This will open up several research directions in terms of how to generate automated tests to activate these security assertions as well as how to utilize these security assertions (properties) for automated property checking.

In Chapter 4, I proposed a scalable test generation approach on RTL models to cover corner cases and rare functional scenarios that are not covered by millions and billions of random tests. It utilizes concolic testing and has two important contributions. (1) I proposed

a directed test generation framework that outperforms the state-of-the-art test generation techniques in activating a single target utilizing contribution-aware edge realignment and effective path selection. (2) I developed two optimization techniques to drastically reduce the overall test generation effort involving multiple targets: (i) target pruning to remove the targets that can be covered by the tests generated for other targets, and (ii) target clustering to minimize the overlapping searches by utilizing learning from related targets. This approach is effective and fast in covering hard-to-detect targets that can be converted to branches.

In Chapter 5, I presented an automated and scalable framework to generate directed tests using concolic testing to activate assertions non-vacuously. This framework first converts any assertion to a branch inside RTL code. Then, it applies concolic testing to generate tests to activate the assertions. While existing model checking based directed test generation can activate assertions, it cannot generate tests for large designs due to state space explosion. Using a diverse set of benchmarks, I have shown that my test generation approach is significantly faster compared to state-of-the-art model checking methods. Most importantly, my approach is scalable since it has linear memory requirement, while state-of-the-art directed test generation methods have exponential memory requirements.

In Chapter 6, I presented quotient space based scalable test generation algorithms that can trade-off between functional coverage and verification effort. My on-the-fly approach uses Euler traversal to cover the whole finite state machine of cache coherence protocol with minimum repeated efforts. This approach utilizes quotient space to group similar transitions together, selects only the representative and important transitions from equivalence classes, and omits only similar transitions to provide scalable test generation framework. My approach is effective on systems with many cores and complex cache coherence protocols, making it suitable for future multicore architectures.

In Chapter 7, I proposed a new paradigm to solve trigger activation problem. This approach is the first attempt in mapping the problem of test generation for trigger activation to the problem of covering maximal satisfiability cliques. I have proved that valid trigger

conditions and satisfiability cliques are one-to-one mapping, and that the test vectors generated by our paradigm are both complete and compact. The presented test generation algorithms repeatedly sample maximal satisfiability cliques and generate a test vector for each of them. It explored the effectiveness of random sampling, lazy construction as well as multi-threading to improve the test generation efficiency. This approach is both scalable and effective in generating efficient test vectors for a wide variety of trigger conditions. This approach can be utilized for activating extremely rare trigger conditions to fulfill diverse requirements such as improvement of functional (trigger) coverage as well as side-channel sensitivity.

In Chapter 8, I proposed a test generation approach to maximize the sensitivity in current-based side-channel analysis. The state-of-the-art test generation technique, e.g., MERS, is not beneficial for large designs due to its high runtime complexity. Most importantly, the sensitivity obtained by the existing approaches is very low compared to environmental noise and process variations, making it useless in practice. I developed an SMT-based first pattern generation algorithm and a genetic algorithm based second pattern generation algorithm that can increase the sensitivity drastically while significantly reduce the test generation time. It breaks down the problem into two sub-problems. The first task generates effective test patterns to maximize the excitation of rare values. The second task finds the best matching pair for each test pattern generated in the first task to maximize the sensitivity. The combination of the SMT-based approach with the genetic algorithm significantly improves both side-channel sensitivity and test generation time, leading to detection of the majority of Trojans under typical process variation and noise margins.

In Chapter 9, I presented an automated test generation approach to take advantage of logic testing in maximizing the difference in path delay. Existing path delay based side-channel analysis techniques are not effective since the difference in path delay between the golden design and Trojan-inserted design is negligible compared to process variation and environmental noise margins. Compared to existing research efforts that fixes one critical path, my approach

explored two different critical paths for the same register in the two designs, resulting in significantly large difference in path delay.

10.2 Future Research Directions

My dissertation proposed many promising approaches for SoC security validation. Considering the increasing importance of SoCs coupled with ever changing landscape of security vulnerabilities, the security validation problem will remain in the forefront in the near future. I will briefly outline how to address future security validation challenges by extending the core principles developed in this dissertation.

I defined security assertions manually by analyzing the specification. Future efforts are needed to analyze the code and specification, and automatically generate security assertions. In addition, current framework only defines assertions for known vulnerabilities. For new/unknown threats in the future, my validation framework can still work as long as the security vulnerabilities can be converted to constraints, and expressed using temporal logic assertions.

Concolic testing is able to address the state explosion problem in formal methods. It explores one path at a time and tries to get "closer" and "closer" to the target based on a simple distance heuristic. For large designs and complex conditions, it may never reach the target due to wrong hint and exploration from the heuristic. Future research can utilize reinforcement learning to learn which new path is better to activate the target. Compared to a distance heuristic, reinforcement learning needs to put more emphasis on the long term rewards.

Validation of cache coherence protocols provides a way to cover only the important states and transitions within a specific debug time. The importance of the transitions are defined using quotient space to group similar transitions together. This approach can be extended to validate the security of caches. Since cache has been exploited by real attacks, such as Spectre and Meltdown, it is crucial to check the existence of side channels and security vulnerabilities.

This can be achieved by generating tests and ensuring that the data is stored or erased as expected.

Finally, the proposed approaches for hardware Trojan detection can increase the probability of activating triggers, maximizing current switching, and maximizing delay sensitivity for unknown Trojans. My evaluations are based on randomly sampled Trojans whose trigger conditions are combinational and are constructed using only rare signals. Future research needs to extend this approach to sequential Trojans.

APPENDIX
LIST OF PUBLICATIONS

Book Chapters:

1. **Yangdi Lyu**, Yuanwen Huang and Prabhat Mishra, SoC Security versus Post-Silicon Debug Conflict, *Post-Silicon Validation and Debug*, P. Mishra and F. Farahmandi (editors), Springer, 2018.

Peer-Reviewed Journal Articles:

1. Subodha Charles, **Yangdi Lyu** and Prabhat Mishra, Real-time Detection and Localization of Distributed DoS Attacks in NoC based SoCs, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2020.

2. **Yangdi Lyu**, Xiaoke Qin, Mingsong Chen and Prabhat Mishra, Directed Test Generation for Validation of Cache Coherence Protocols, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

3. **Yangdi Lyu** and Prabhat Mishra, A Survey of Side-Channel Attacks on Caches and Countermeasures, *Journal of Hardware and Systems Security (HaSS)*, 2017.

Peer-Reviewed Conference Papers:

1. **Yangdi Lyu** and Prabhat Mishra, Directed Test Generation for Delay-based Side Channel Analysis, *Design Automation and Test in Europe (DATE)*, 2020.

2. **Yangdi Lyu** and Prabhat Mishra, Automated Trigger Activation by Repeated Maximal Clique Sampling, *Asia and South Pacific Design Automation Conference*, 2020.

3. **Yangdi Lyu** and Prabhat Mishra, Automated Test Generation for Activation of Assertions in RTL Models, *Asia and South Pacific Design Automation Conference*, 2020.

4. **Yangdi Lyu** and Prabhat Mishra, Efficient Test Generation for Trojan Detection using Side Channel Analysis, *Design Automation and Test in Europe (DATE)*, 2019.

5. **Yangdi Lyu**, Alif Ahmed and Prabhat Mishra, Automated Activation of Multiple Targets in RTL Models using Concolic Testing, *DATE*, 2019. **Best paper nomination**

6. Subodha Charles, **Yangdi Lyu** and Prabhat Mishra, Real-time Detection and Localization of DoS Attacks in NoC based SoCs, *DATE*, 2019.

7. **Yangdi Lyu** and Alper Üngör, A Fast 2- Approximation Algorithm for Guarding Orthogonal Terrains, *In Proc. of 28th Canadian Conf. on Comp. Geometry (CCCG)*, 2016.

Patents and Copyrights:

1. Prabhat Mishra, and **Yangdi Lyu**, Delay-based side-channel analysis for Trojan detection, U.S. Provisional Patent Application Serial No. 62/966,657, filed Jan 28, 2020.

2. Prabhat Mishra, Subodha Charles and **Yangdi Lyu**, Securing System-on-Chip using Incremental Cryptography, U.S. Provisional Patent Application Serial No. 62/874,187, filed July 15, 2019.

3. Prabhat Mishra and **Yangdi Lyu**, Maximization of Side-Channel Sensitivity for Trojan Detection, U.S. Provisional Patent Application Serial No. 62/869,288, filed July 1, 2019.

4. Prabhat Mishra and **Yangdi Lyu**, Trigger Activation by Repeated Maximal Clique Sampling, U.S. Provisional Patent Application Serial No. 62/869,294, filed July 1, 2019.

5. Prabhat Mishra, Subodha Charles and **Yangdi Lyu**, Real-Time Detection and Localization of DoS Attacks in NoC based SoC Architectures, U.S. Provisional Patent Application Serial No. 62/868,258, filed June 28, 2019.

REFERENCES

[1] A. Pakonen, C. Pang, I. Buzhinsky, and V. Vyatkin, "User-friendly formal specification languages-conclusions drawn from industrial experience on model checking," in *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE, 2016, pp. 1–8.

[2] A. Ahmed and P. Mishra, "Quebs: Qualifying event based search in concolic testing for validation of rtl models," in *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 185–192.

[3] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on rtl models," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 1538–1543.

[4] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *2015 IEEE Computer Society Annual Symposium on VLSI*, July 2015, pp. 7–12.

[5] Y. Lyu and P. Mishra, "Efficient test generation for trojan detection using side channel analysis," in *Design Automation and Test in Europe (DATE), Florence, Italy, March 25 - 29*, 2019.

[6] Y. Huang, S. Bhunia, and P. Mishra, "Scalable test generation for trojan detection using side channel analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 11, pp. 2746–2760, Nov 2018.

[7] L. Liu, E. G. Larsson, W. Yu, P. Popovski, C. Stefanovic, and E. de Carvalho, "Sparse signal processing for grant-free massive connectivity: A future paradigm for random access protocols in the internet of things," *IEEE Signal Processing Magazine*, vol. 35, no. 5, pp. 88–99, Sep. 2018.

[8] L. Salmon, "System security integrated through hardware and firmware (ssith)," 2017, https://www.darpa.mil/attachments/SSITHProposersDay20170422.pdf.

[9] F. Farahmandi, "Formal verification of hardware security and trust," Ph.D. dissertation, University of Florida, 2018.

[10] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-based design*. Springer Science & Business Media, 2004.

[11] I. Wagner and V. Bertacco, "Mcjammer: adaptive verification for multi-core designs," in *Proc of DATE*, 2008, pp. 670–675.

[12] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *Cryptographic Hardware and Embedded Systems - CHES 2009*, C. Clavier and K. Gaj, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 396–410.

[13] D. Ismari, J. Plusquellic, C. Lamech, S. Bhunia, and F. Saqib, "On detecting delay anomalies introduced by hardware trojans," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2016, pp. 1–7.

[14] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 952–965, June 2013.

[15] J. Li and J. Lach, "Negative-skewed shadow registers for at-speed delay variation characterization," in *2007 25th International Conference on Computer Design*, Oct 2007, pp. 354–359.

[16] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[17] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[18] P. Mishra, R. Morad, A. Ziv, and S. Ray, "Post-silicon validation in the soc era: A tutorial introduction," *IEEE Design & Test*, vol. 34, pp. 68–92, 2017.

[19] M. Chen, X. Qin, H.-M. Koo, and P. Mishra, *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*. Springer Publishing Company, Incorporated, 2012.

[20] "National vulnerability database," https://nvd.nist.gov.

[21] "AMD64 Architecture Programmer's Manual Volume 2: System Programming," http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf, 2013.

[22] M. E. Thomadakis, "The architecture of the Nehalem processor and Nehalem-EP SMP platforms," *Resource*, vol. 3, p. 2, 2011.

[23] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, Jan 2010.

[24] M. Chen, X. Qin, H.-M. Koo, and P. Mishra, *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*, 1st ed. Springer Publishing Company, Incorporated, 2012.

[25] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware trojan attacks: Threat analysis and countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, Aug 2014.

[26] N. Jacob, D. Merli, J. Heyszl, and G. Sigl, "Hardware trojans: current challenges and approaches," *IET Computers Digital Techniques*, vol. 8, no. 6, pp. 264–273, 2014.

[27] S. Ghosh, A. Basak, and S. Bhunia, "How secure are printed circuit boards against trojan attacks?" *IEEE Design Test*, vol. 32, no. 2, pp. 7–16, April 2015.

[28] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, pp. 6:1–6:23, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2906147

[29] F. Courbon, P. Loubet-Moundi, J. J. A. Fournier, and A. Tria, "A high efficiency hardware trojan detection technique based on fast sem imaging," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 788–793.

[30] "ISCAS85 combinational benchmark circuits," https://filebox.ece.vt.edu/~mhsiao/iscas85.html.

[31] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 146–162, Oct. 1999.

[32] M. Chen and P. Mishra, "Property learning techniques for efficient generation of directed tests," *IEEE Transactions on Computers*, vol. 60, no. 6, pp. 852–864, June 2011.

[33] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095430.1081750

[34] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065036

[35] X. Qin and P. Mishra, "Automated generation of directed tests for transition coverage in cache coherence protocols," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2012, pp. 3–8.

[36] Y. C. Randal E. Bryant, "Verification of arithmetic circuits with binary moment diagrams," in *32nd Design Automation Conference*, 1995, pp. 535–541.

[37] M. J. Ciesielski, P. Kalla, Zhihong Zheng, and B. Rouzeyre, "Taylor expansion diagrams: a compact, canonical representation with applications to symbolic verification," in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002, pp. 285–289.

[38] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems*, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.

[39] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas, "Effective theorem proving for hardware verification," in *Theorem Provers in Circuit Design*, R. Kumar and T. Kropf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 203–222.

[40] D. Wood, G. Gibson, and R. Katz, "Verifying a multiprocessor cache controller using random test generation," *IEEE Design Test of Computers*, vol. 7, no. 4, pp. 13 –25, 1990.

[41] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-pro: innovations in test program generation for functional processor verification," *IEEE Design Test of Computers*, vol. 21, no. 2, pp. 84 – 93, 2004.

[42] D. Abts, S. Scott, and D. Lilja, "So many states, so little time: verifying memory coherence in the Cray X1," in *Proc of ISPDP*, 2003.

[43] Y. Oddos, K. Morin-Allory, D. Borrione, M. Boulé, and Z. Zilic, "Mygen: Automata-based on-line test generator for assertion-based verification," in *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '09. New York, NY, USA: ACM, 2009, pp. 75–80. [Online]. Available: http://doi.acm.org/10.1145/1531542.1531563

[44] J. G. Tong, M. Boulé, and Z. Zilic, "Test compaction techniques for assertion-based test generation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 19, no. 1, pp. 9:1–9:29, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534397

[45] M. Chen, X. Qin, and P. Mishra, "Efficient decision ordering techniques for sat-based test generation," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 490–495.

[46] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142 – 170, 1992.

[47] E. Clarke, O. Grumberg and D. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

[48] D. Dill, A. Drexler, A. Hu, and C. Yang, "Protocol verification as a hardware design aid," in *Proc of ICCD*, 1992, pp. 522 –525.

[49] J. Whittemore, J. Kim, and K. Sakallah, "Satire: A new incremental satisfiability engine," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, June 2001, pp. 542–545.

[50] O. Strichman, "Accelerating bounded model checking of safety properties," *Formal Methods in System Design*, vol. 24, no. 1, pp. 5–24, Jan 2004.

[51] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards trojan-free trusted ics: Problem analysis and detection scheme," in *2008 Design, Automation and Test in Europe*, March 2008, pp. 1362–1365.

[52] A. Waksman, M. Suozzo, and S. Sethumadhavan, "Fanci: Identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 697–708. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516654

[53] T. F. Wu, K. Ganesan, Y. A. Hu, H. . P. Wong, S. Wong, and S. Mitra, "Tpad: Hardware trojan prevention and detection for trusted integrated circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 4, pp. 521–534, April 2016.

[54] A. Bazzazi, M. T. Manzuri Shalmani, and A. M. Hemmatyar, "Hardware trojan detection based on logical testing," *J. Electron. Test.*, vol. 33, no. 4, pp. 381–395, Aug. 2017. [Online]. Available: https://doi.org/10.1007/s10836-017-5670-0

[55] M. E. Amyeen, S. Venkataraman, A. Ojha, and S. Lee, "Evaluation of the quality of n-detect scan atpg patterns on a processor," in *2004 International Conference on Test*, Oct 2004, pp. 669–678.

[56] "ISCAS89 sequential benchmark circuits," https://filebox.ece.vt.edu/~mhsiao/iscas89.html.

[57] M. A. Nourian, M. Fazeli, and D. Hely, "Hardware trojan detection using an advised genetic algorithm based logic testing," *Journal of Electronic Testing*, vol. 34, no. 4, pp. 461–470, Aug 2018. [Online]. Available: https://doi.org/10.1007/s10836-018-5739-4

[58] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[59] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 443–446.

[60] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated software debugging," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 321–334. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755946

[61] F. Charreteur and A. Gotlieb, "Constraint-based test input generation for java bytecode," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 131–140.

[62] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 363–374, 2009.

[63] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 31–36.

[64] K. Cong, F. Xie, and L. Lei, "Automatic concolic test generation with virtual prototypes for post-silicon validation," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2013, pp. 303–310.

[65] B. Chen, K. Cong, Z. Yang, Q. Wang, J. Wang, L. Lei, and F. Xie, "End-to-end concolic testing for hardware/software co-validation," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, June 2019, pp. 1–8.

[66] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "Crete: A versatile binary-level concolic testing framework," in *Fundamental Approaches to Software Engineering*, A. Russo and A. Schürr, Eds. Cham: Springer International Publishing, 2018, pp. 281–298.

[67] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A risc-v case study*," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019, pp. 1–6.

[68] B. Lin, K. Cong, Z. Yang, Z. Liao, T. Zhan, C. Havlicek, and F. Xie, "Concolic testing of systemc designs," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, March 2018, pp. 1–7.

[69] L. Liu and S. Vasudevan, "Star: Generating input vectors for design validation by static analysis of rtl," in *2009 IEEE International High Level Design Validation and Test Workshop*, Nov 2009, pp. 32–37.

[70] ——, "Scaling input stimulus generation through hybrid static and dynamic analysis of rtl," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 1, p. 4, 2014.

[71] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using ic fingerprinting," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, May 2007, pp. 296–310.

[72] Y. Jin and Y. Makris, "Hardware trojan detection using path delay fingerprint," in *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, June 2008, pp. 51–57.

[73] R. Rad, J. Plusquellic, and M. Tehranipoor, "A sensitivity analysis of power signal methods for detecting hardware trojans under real process and environmental conditions," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 12, pp. 1735–1744, DECEMBER 2010.

[74] H. Salmani, M. Tehranipoor, and J. Plusquellic, "A novel technique for improving hardware trojan detection and reducing trojan activation time," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 1, pp. 112–125, Jan 2012.

[75] A. N. Nowroz, K. Hu, F. Koushanfar, and S. Reda, "Novel techniques for high-sensitivity hardware trojan detection using thermal and power maps," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 12, pp. 1792–1805, Dec 2014.

[76] S. Narasimhan, D. Du, R. S. Chakraborty, S. Paul, F. G. Wolff, C. A. Papachristou, K. Roy, and S. Bhunia, "Hardware trojan detection by multiple-parameter side-channel analysis," *IEEE Transactions on Computers*, vol. 62, no. 11, pp. 2183–2195, Nov 2013.

[77] C. Bao, D. Forte, and A. Srivastava, "Temperature tracking: Toward robust run-time detection of hardware trojans," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1577–1585, Oct 2015.

[78] S. K. Rao, D. Krishnankutty, R. Robucci, N. Banerjee, and C. Patel, "Post-layout estimation of side-channel power supply signatures," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, May 2015, pp. 92–95.

[79] J. Plusquellic and F. Saqib, *Detecting Hardware Trojans Using Delay Analysis*. Cham: Springer International Publishing, 2018, pp. 219–267.

[80] Y. Huang, S. Bhunia, and P. 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*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 130–141. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978396

[81] B. Balaji, J. McCullough, R. K. Gupta, and Y. Agarwal, "Accurate characterization of the variability in power consumption in modern mobile processors," in *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*. Hollywood, CA: USENIX, 2012.

[82] F. Farahmandi, R. Morad, A. Ziv, Z. Nevo, and P. Mishra, "Cost-effective analysis of post-silicon functional coverage events," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 392–397.

[83] M. Ben-Ari, A. Pnueli, and Z. Manna, "The temporal logic of branching time," *Acta informatica*, vol. 20, no. 3, pp. 207–226, 1983.

[84] A. Pnueli, "The temporal logic of programs," in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, oct 1977, pp. 46–57.

[85] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, D. Kozen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71.

[86] G. Di Guglielmo, L. Di Guglielmo, A. Foltinek, M. Fujita, F. Fummi, C. Marconcini, and G. Pravadelli, "On the integration of model-driven design and dynamic assertion-based verification for embedded software," *Journal of Systems and Software*, vol. 86, no. 8, 2013.

[87] "1850-2010 - IEEE Standard for Property Specification Language (PSL)," 2010.

[88] "1800-2012 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," 2012.

[89] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer *et al.*, "The forspec temporal logic: A new temporal property-specification language," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2002, pp. 296–311.

[90] A. Bauer and M. Leucker, "The theory and practice of salt," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 13–40.

[91] D. Tabakov, G. Kamhi, M. Y. Vardi, and E. Singerman, "A temporal language for systemc," in *2008 Formal Methods in Computer-Aided Design*, Nov 2008, pp. 1–9.

[92] H. Foster, K. Larsen, and M. Turpin, "Introduction to the new accellera open verification library," in *DVCon'06: Proceedings of the Design and Verification Conference and exhibition*. Citeseer, 2006.

[93] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke, "Automatic generation of complex properties for hardware designs," in *2008 Design, Automation and Test in Europe*, March 2008, pp. 545–548.

[94] "Common weakness enumeration," https://cwe.mitre.org/.

[95] "Arm trustzone," https://developer.arm.com/technologies/trustzone.

[96] "Opencores," https://www.opencores.org/.

[97] "Intel® software guard extensions," https://software.intel.com/en-us/sgx.

[98] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra, "Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution," in *IEEE International Test Conference (ITC)*, 2018.

[99] Y. Lyu and P. Mishra, "Automated trigger activation by repeated maximal clique sampling," in *Asia and South Pacific Design Automation Conference (ASPDAC), Beijing, China, January 13 - 16*, 2020.

[100] D. Kroening and M. Purandare, *EBMC: The Enhanced Bounded Model Checker*, http://www.cprover.org/ebmc.

[101] S. Williams, "Icarus verilog," *On-line: http://iverilog.icarus.com/*, 2006.

[102] "Edautils website," http://www.edautils.com.

[103] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 737–744.

[104] F. Corno, M. S. Reorda, and G. Squillero, "Rt-level itc'99 benchmarks and first atpg results," *IEEE Design Test of Computers*, vol. 17, no. 3, pp. 44–53, July 2000.

[105] "Trusthub," https://www.trust-hub.org/, accessed: 2018-10-10.

[106] L. Ferro, L. Pierre, Y. Ledru, and L. du Bousquet, "Generation of test programs for the assertion-based verification of tlm models," in *2008 3rd International Design and Test Workshop*, Dec 2008, pp. 237–242.

[107] B. Pal, A. Banerjee, A. Sinha, and P. Dasgupta, "Accelerating assertion coverage with adaptive testbenches," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 5, pp. 967–972, May 2008.

[108] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in temporal model checking," *Formal Methods in System Design*, vol. 18, no. 2, pp. 141–163, Mar 2001. [Online]. Available: https://doi.org/10.1023/A:1008779610539

[109] S. Williams, "Icarus verilog," http://iverilog.icarus.com.

[110] P. Mishra and N. Dutt, "Graph-based functional test program generation for pipelined processors," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, Feb 2004, pp. 182–187.

[111] J. Edmonds and E. L. Johnson, "Matching, euler tours and the chinese postman," *Mathematical Programming*, vol. 5, no. 1, pp. 88–124, 1973.

[112] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[113] G. Zhang, W. Horn, and D. Sanchez, "Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 13–25.

[114] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[115] R. M. Karp, *Reducibility among Combinatorial Problems*. Boston, MA: Springer US, 1972, pp. 85–103. [Online]. Available: https://doi.org/10.1007/978-1-4684-2001-2_9

[116] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo, *The Maximum Clique Problem*. Boston, MA: Springer US, 1999, pp. 1–74.

[117] J. W. Moon and L. Moser, "On cliques in graphs," *Israel Journal of Mathematics*, vol. 3, no. 1, pp. 23–28, Mar 1965. [Online]. Available: https://doi.org/10.1007/BF02760024

[118] C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *Commun. ACM*, vol. 16, no. 9, pp. 575–577, Sep. 1973.

[119] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova, "Genome-scale computational approaches to memory-intensive applications in systems biology," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 12–.

[120] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park, "A scalable, parallel algorithm for maximal clique enumeration," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 417 – 428, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731509000082

[121] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[122] E. L. Lawler, J. Karel Lenstra, and A. H. G. Rinnooy Kan, "Generating all maximal independent sets: Np-hardness and polynomial-time algorithms," *SIAM J. Comput.*, vol. 9, pp. 558–565, 08 1980.

[123] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theoretical Computer Science*, vol. 363, no. 1, pp. 28 – 42, 2006, computing and Combinatorics. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397506003586

[124] "Saededk90core - 90nm digital standard cell li-brary," http://www.synopsys.com/community/universityprogram/pages/library.aspx.

BIOGRAPHICAL SKETCH

Yangdi Lyu received his Ph.D. degree in the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, USA in 2020. He received his B.E. degree in Department of Hydraulic Engineering from Tsinghua University, Beijing, China in 2011. His research interests include system-on-chip verification, hardware security validation, and side channel attacks. He has published one book chapter, three journal papers and seven conference papers. He has served as a reviewer of several premier international conferences and journals.