THE UNIVERSITY OF CHICAGO

AUTOMATED LOCALIZATION OF TIMING-RELATED SECURITY BUGS IN HARDWARE DESIGNS

A THESIS SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER'S IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
TROY HU

CHICAGO, ILLINOIS
JUNE, 2020

TABLE OF CONTENTS

LIST OF FIGURES vi

LIST OF TABLES vii

ACKNOWLEDGMENTS viii

ABSTRACT ix

1 INTRODUCTION 1

2 MOTIVATION, BACKGROUND, AND RELATED WORK 4
- 2.1 Cache Side-Channel Attacks 4
  - 2.1.1 Flush-based Attacks 4
  - 2.1.2 Conflict-based attacks 5
  - 2.1.3 New Attacks 6
- 2.2 Cache Attack Applications 7
  - 2.2.1 Cache-based Exploits 7
  - 2.2.2 Out-of-Order and Speculative Execution Exploits 7
- 2.3 Hardware-Based Cache Side-Channel Mitigations 8
  - 2.3.1 Complete Cache Access Avoidance 8
  - 2.3.2 Eviction Set Creation Prevention 9
  - 2.3.3 Victim Cache Access Inference Prevention 9
- 2.4 Other Insecure Hardware 10
- 2.5 Information Flow Tracking 11
  - 2.5.1 Timing vs. Functional Flows 11
  - 2.5.2 Single Label IFT Techniques 12
  - 2.5.3 Multiple Label IFT Techniques 12
  - 2.5.4 Using IFT to Ensure Secure Hardware Designs 15
  - 2.5.5 Bug Localization 17
- 2.6 Research Problem and Goals 18

3 DEFINITIONS AND ASSUMPTIONS 20
- 3.1 Assumptions 20
- 3.2 Definitions 20

4 SELECTIVE TAINT GENERATION 23
- 4.1 Detailed Description 23
- 4.2 Accuracy Evaluation 24
- 4.3 Efficiency Evaluation 25
  - 4.3.1 Time Complexity 26
  - 4.3.2 Space Complexity 26

5 TAINT GENERATION LOGGING TECHNIQUE . . . 27
5.1 Detailed Description . . . 27
5.1.1 Pre-Processing . . . 28
5.1.2 Simulation and Candidate Identification . . . 28
5.2 Experimental Methodology . . . 30
5.2.1 Validating Candidate Set . . . 31
5.2.2 Multiplier . . . 31
5.2.3 Divider . . . 31
5.2.4 AES Cores Injected with Divider . . . 32
5.2.5 RSA Core . . . 33
5.2.6 Direct Mapped Cache . . . 34
5.2.7 Experimental Setup . . . 35
5.3 Accuracy Evaluation . . . 35
5.3.1 Multiplier . . . 35
5.3.2 Divider . . . 36
5.3.3 AES Cores Injected with Divider - Base Example . . . 37
5.3.4 AES Cores Injected with Divider - Loop Example . . . 38
5.3.5 AES Cores Injected with Divider - Timing Blockage Example . . . 39
5.3.6 RSA . . . 40
5.3.7 Cache . . . 41
5.3.8 Experiments' Summary . . . 41
5.3.9 Accuracy Analysis on Handmade Examples . . . 41
5.4 Efficiency Evaluation . . . 44
5.4.1 Time Complexity . . . 45
5.4.2 Space Complexity . . . 45

6 BRUTE-FORCE BACKTRACKING . . . 46
6.1 Description . . . 46
6.2 Accuracy Evaluation . . . 47
6.3 Efficiency Evaluation . . . 47
6.3.1 Time Complexity . . . 48
6.3.2 Space Complexity . . . 48

7 AGGREGATED GRAPH WALKING . . . 50
7.1 Description . . . 50
7.1.1 Graphing . . . 50
7.1.2 Candidate Precision Improvement . . . 52
7.2 Experimental Methodology . . . 52
7.3 Accuracy Evaluation . . . 53
7.3.1 Multiplier . . . 53
7.3.2 Divider . . . 54
7.3.3 AES Cores Injected with Divider - Base Example . . . 54
7.3.4 AES Cores Injected with Divider - Loop Example . . . 55
7.3.5 AES Cores Injected with Divider - Timing Blockage Example . . . 56
7.3.6 RSA . . . 56

7.3.7 Cache . . . 57
7.3.8 Experiments' Summary . . . 58
7.3.9 Accuracy Analysis on Handmade Examples . . . 58
7.4 Efficiency Evaluation . . . 61
7.4.1 Time Complexity . . . 62
7.4.2 Space Complexity . . . 63

8 UNIQUE TAINT TRACKING . . . 64
8.1 Description . . . 64
8.2 Accuracy Evaluation . . . 66
8.3 Efficiency Evaluation . . . 66
8.3.1 Time Complexity . . . 67
8.3.2 Space Complexity . . . 68

9 THE BLOOM FILTER APPROACH . . . 69
9.1 Description . . . 70
9.2 Accuracy Evaluation . . . 72
9.3 Efficiency Evaluation . . . 75
9.3.1 Time Complexity . . . 75
9.3.2 Space Complexity . . . 76

10 PROBABILISTIC PATHING . . . 77
10.1 Description . . . 77
10.2 Accuracy Evaluation . . . 78
10.3 Efficiency Evaluation . . . 81
10.3.1 Time Complexity . . . 82
10.3.2 Space Complexity . . . 82

11 SUMMARIES AND COMPARISONS OF TECHNIQUES . . . 83
11.1 Technique Summaries . . . 83
11.2 Technique Analyses and Comparisons . . . 86

12 OTHER DISCUSSION . . . 90
12.1 Optimal Number of Faulty Simulations . . . 90
12.2 Using the Techniques to Fix Bugs . . . 90
12.3 Technique Automation . . . 91

13 FUTURE WORKS AND CONCLUSION . . . 92
13.1 Future Works . . . 92
13.1.1 Implementation and Empirical Evaluation of the Techniques . . . 92
13.1.2 Complexity and Size of Empirically Tested Designs . . . 92
13.1.3 Technique Optimizations . . . 93
13.1.4 Non-Timing Security Bugs (Functional Flows) . . . 93
13.2 Conclusion . . . 94

LIST OF FIGURES

2.1 Procedural block of a divider module that generates and blocks sensitive timing flows [5]. This example is similar to the pseudo code example found in Figure 1 of the Clepsydra paper [1]. Our analysis of this example is also based on the Clepsydra paper's explanation of its own example. 13

5.1 A faulty simulation on a theoretical design that results in disjointed register false positives in the Taint Generation Logging Technique's candidate set. Each rounded rectangle represents a register and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box. 42
5.2 Two faulty simulations (a) and (b) on another theoretical design that result in disjointed register false positives in the Taint Generation Logging Technique's candidate set. Each rounded rectangle represents a register and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box. 43

7.1 The multiplier example's register dependency graph. 53
7.2 The divider example's register dependency graph. 54
7.3 The AES with divider base example's register dependency graph (truncated). . . 55
7.4 The AES with divider - Loop example's register dependency graph (truncated). 55
7.5 RSA example's register dependency graph. 57
7.6 Cache example's register dependency graph. 57
7.7 A faulty simulation on a theoretical design that results in disjointed register false positives in the AGW Technique's candidate set. Each rounded rectangle represents a register and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box. 59
7.8 Dependency graph produced from the example in Figure 8.3 59
7.9 Two faulty simulations (a) and (b) on a theoretical design with blockage points that result in disjointed register false positives in the AGW Technique's candidate set. Each rounded rectangle represents a register and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box. 60
7.10 Dependency graph produced from the example in Figure 8.5 60

LIST OF TABLES

5.1 The IFT timing behavior of monitored label affecting registers in the multiplier example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow. 36

5.2 The IFT timing behavior of monitored label affecting registers in the divider example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow. 36

5.3 The IFT timing behavior of monitored label affecting registers in the AES with divider - base example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow. 37

5.4 The IFT timing behavior of monitored label affecting registers in the AES with divider - Loop example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow. 38

5.5 The IFT timing behavior of monitored label affecting registers in the AES with divider - Timing Blockage example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow. 39

5.6 The IFT timing behavior of monitored label affecting registers in the RSA example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow. 40

5.7 The IFT timing behavior of monitored label affecting registers in the cache example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow. 41

11.1 A summary of the accuracy and informal worst-case computational complexities of each technique. 83

11.2 A summary of the main strengths and weaknesses of each technique. 84

ACKNOWLEDGMENTS

First and foremost, I thank my advisor Professor Yanjing Li for her help and guidance over the past two years. She has been instrumental in my development as a computer scientist/hardware researcher and I cannot stress enough that I would not be here without her.

I also thank Armita Ardeshiricham and Ryan Kastner for providing help and insights relating to information flow tracking and hardware security. They taught me a lot about hardware verification and provided me with the tools and IFT code I needed for my experiments.

I thank Professor Shan Lu for taking the time to serve as a member of my committee.

I thank Olivia Weng and Henry He for helping me with understanding concepts about hardware security and setting up my experiments.

Finally, I thank all of my friends and family who have supported me these past four years. Your help is invaluable.

ABSTRACT

Hardware systems today are largely at risk of leaking sensitive data through timing variations in low-level operations, which leads to severe consequences such as confidentiality breaches and malicious actors gaining root privileges. For example, the famous Spectre and Meltdown attacks on modern processors are possible partly due to timing variations in the caches. In order to ensure timing secure designs, hardware designers must be provided with the necessary tools to detect and localize the bugs. While there exist techniques that can detect timing-related security bugs, e.g. Information Flow Tracking (IFT), bug localization is primarily performed through manual efforts, which are time consuming and inefficient. In this paper, we present seven techniques that can automatically localize timing-related security bugs in hardware designs. The key idea of these techniques involves the analysis of timing behaviors in fault-causing simulations. We can obtain these behaviors by leveraging existing IFT infrastructure with low cost and high precision. Once we present these techniques, we evaluate their accuracies through various methods ranging from empirical experiments to analyses on theoretical designs. Furthermore, we informally analyze their efficiencies. Finally, we leave the further elaboration, implementation, extension, and empirical evaluation of these techniques as novel directions for future work in security bug localization.

CHAPTER 1
INTRODUCTION

Most, if not all, of today's systems contain hardware that suffers or has the potential to suffer from timing-related security bugs. A component of hardware possesses a timing-related security bug if one of its operations exhibits variations in execution time that depend on the value of sensitive data. These variations are observable by the outside world and thus break the confidentiality property of security; potentially untrusted entities can observe these variations to infer the values of sensitive data. Timing bugs are subtle and often overlooked in the hardware design process. The lack of attention towards these bugs leaves a system vulnerable to a multitude of dangerous exploits such as those that target the cache.

One of the most prominent hardware examples that contains timing-related security bugs and is a major source of motivation for this paper is the cache. In the cache, sensitive data leakage often comes from timing and access-based side-channel exploits. They usually leak data from a victim process, e.g. a cryptographic algorithm's secret keys [8, 13, 19, 20, 24, 26, 37, 38], by taking advantage of timing variations in the cache (e.g. cache accesses). Cache attacks are especially dangerous since they are diverse and efficient. For example, some techniques can take advantage of cache directories [37] while others can exploit transactional aborts to quickly leak data [6]. Cache attacks can not only be utilized standalone, but also to augment other attacks. Out-of-order and speculative execution exploits such as Meltdown [18] and Spectre [15] use cache attacks to leak data to an attacking process. It is therefore imperative that exploit mitigations be implemented in all caches.

The hardware-based mitigations for cache attacks come in a multitude of forms. The Partition-Locked Cache (PLCache) [34], for example, mitigates side-channels by allowing processes to dynamically partition the cache into secure regions. In order for a hardware designer to practically design and then implement fully secure mitigations, they need the ability verify them for security. Without automated verification and debugging tools, it is difficult for designers to quickly, efficiently, and securely construct mitigations, especially more complex ones. Without these tools, designers waste resources and may even unknowingly make mistakes during the design or implementation phase. For example, Ardeshiricham et al. [3] recently formally analyzed a PLCache design and found a timing-related side-channel in the process. Moreover, while caches are a main source of timing-based security bugs, any hardware that possesses timing variations in one of its operations has the potential to be buggy. With so many places in hardware for sensitive data to leak through timing differences, it is extremely necessary to provide hardware designers with tools to verify and debug their designs for timing-based security bugs. One tool that can be effectively used to detect such security bugs is called Information Flow Tracking (IFT).

Security flaws within hardware should ideally be caught in the design phase (pre-silicon). IFT is a pre-silicon security debugging technique that can detect security bugs by tracking a datum's sensitivity level as it flows through a hardware design. Information flows can be tracked at different levels of hardware, e.g. gate level [32] or Register-Transfer Level (RTL) [2, 1, 3]. Some strategies, such as Clepsydra [1] and VeriSketch [3], have developed IFT rules that separate information flows into timing-based flows (whether the availability of some data contains timing variations [23]), and functional flows (whether some data depends on other data [23]). However, while IFT techniques can be used to automatically detect security flaws, to the best of our knowledge, they have not been used to automatically find the origin of, or localize, such flaws. Moreover, as far as we know, automated timing-related security bug localization in hardware designs has not been thoroughly explored. The hardware designer, who must fix the bug, is therefore left with the arduous and not necessarily guaranteed task of manually finding the bug's location. The patching of timing-related security bugs is therefore difficult and likely impractical for the programmer. We now present our solution to this problem (thesis statement).

Thesis Statement: The behaviors of IFT can be utilized to accurately localize timing-based hardware security design bugs. Thus, we develop a host of IFT-based bug localization techniques with varying degrees of accuracy and efficiency.

In this paper, we seek to specifically utilize VeriSketch's and Clepsydra's IFT strategy to automatically identify the location of a detected bug. We present and explore seven techniques that can automatically localize timing-related security bugs in Verilog hardware designs by analyzing the IFT behaviors of security violating simulations. To the best of our knowledge, this is the first work in the IFT space that attempts to automatically localize timing-related security related bugs in hardware designs. We examine each strategy's effectiveness, both in terms of accuracy and efficiency, through various methods ranging from empirical evaluations to logical reasoning to informal complexity analyses.

In Chapter 2, we discuss the background, related work, and motivation for this paper. Then in Chapter 3, we detail the assumptions and definitions that our techniques operate under. In Chapters 4 through 10, we describe, evaluate, and discuss the seven techniques we consider. In Chapter 11, we summarize and compare the techniques' accuracies and efficiencies. In Chapter 12, we discuss other technique-related topics. Finally in Chapter 13, we discuss future works for these strategies and conclude this paper.

CHAPTER 2
MOTIVATION, BACKGROUND, AND RELATED WORK

Timing-based security bugs occur when sensitive data is exposed, or leaks, to other and potentially malicious processes as a result of data dependent timing variations in a system's execution. Many of today's hardware components' execution time, such as cache accesses, depend on the values of some data and are thus vulnerable to having timing bugs. As these vulnerabilities rely on time, they are harder to track than traditional bugs, requiring special techniques to detect or prevent them. These timing bugs are therefore often overlooked during the hardware design process. We begin this chapter by motivating the need for timing-based security debugging tools by examining a multitude of timing-based security bugs in hardware designs. We then discuss approaches that can be used to detect security bugs. Finally, we examine and highlight the limitations of current security bug prevention research and bug localization techniques to motivate the need for our localization techniques.

2.1 Cache Side-Channel Attacks

In most cache side-channel attacks, the attacker monitors a victim's cache lines or sets for data access latency differences induced by evictions or conflicts. The attacker can then utilize these behavioral observations to infer the victim's cache accesses and in turn secret information. In this section, we provide background for and detailed descriptions of these cache attacks.

2.1.1 Flush-based Attacks

Flush+Reload [38] is a flushed-based attack that targets Last Level Caches (LLCs) through the use of shared memory and cache flushing instructions, i.e. `clflush`, to monitor a victim's cache accesses. Before the attack begins, the attacker loads into its own address space the victim's shared libraries or executable. Since pages of memory are shared among different processes, the attacker obtains direct access to the victim data's or instructions' locations in the cache. After the loading step, the attacker only needs to perform the following operations:

1. *Flush* - Use flushing instructions to directly flush all monitored cache lines.

2. *Wait* - Wait.

3. *Reload* - Reload each of the monitored cache lines and time their reload latencies.

If the attacker observes a high reload latency for all of the cache lines, they infer that the victim did not access those lines during the wait period. On the other hand, a low reload latency for a cache line indicates to the attacker the victim's access to that line. The attacker can then use this knowledge to infer secret information.

Flush+Flush [8] is a similar attack to Flush+Reload. It exploits shared memory and utilizes flush instructions to evict specific cache lines. However, Flush+Flush discards the reload step (Step 3) of Flush+Reload. Instead, Flush+Flush continuously flushes a victim's cache line. A quick flush tells the attacker that the victim did not recently access the line. A slow flush suggests that the victim recently accessed the line and loaded it into the cache.

2.1.2 Conflict-based attacks

Unlike flush-based attacks which utilize cache flushing instructions, conflict-based attacks rely on causing cache conflicts with victim lines to leak sensitive information. Percival's Attack [26] is a simple conflict-based cache attack that constructs an appropriately spaced array and then continuously accesses the array's elements in sequence. Depending on the access latency of an element corresponding to a cache set, the attacker can deduce whether the victim process accessed that set. Evict+Time [24] is also a conflict-based attack that spies on a victim's cache set accesses by evicting specific sets from the cache and then timing the entire run of a victim application.

Prime+Probe [24] is a popular conflict-based attack that targets the L1 cache and with some modifications, e.g. through the use of large pages, the LLC as well [13, 20]. It spies on the victim's cache accesses through the use of eviction sets. Eviction sets are "groups of virtual addresses that map to the same cache set" [33, p. 1]. Vila et al. [33] note that these sets must be big enough so that an access to a subset of its addresses will evict an entire cache set. After the attacker finds an eviction set for some of the victim's data, they only need to:

1. *Prime* - Replace all of the lines in the victim's data's cache set with the eviction set.
2. *Wait* - Wait.
3. *Probe* - Reload the eviction set and time the accesses.

A high memory access latency for one of the lines, i.e. a miss, in the eviction set indicates to the attacker of the victim's access to that cache set during the wait period. Otherwise, the attacker will incur a cache hit for all of the lines and determine that the victim did not access that set.

2.1.3 New Attacks

Yan et al. [37] observed that many cache attacks have been rendered ineffective by the increasing popularity of non-inclusive and/or sliced caches. They explained that many attacks ensure a victim's data's absence from the entire cache after the conflict step by utilizing cache inclusivity. If a cache is non-inclusive, there is no longer such a guarantee, leaving the attacker with no precise way to know whether the victim process accessed a monitored cache line/set. The researchers also observed that sliced cache designs introduce levels of indirection in the mapping between address and cache location, thus impeding the eviction set creation process. In the same paper, Yan et al. proposed an attack that is able to overcome these problems by attacking inclusive cache directories instead. This attack discovers the cache directory structure with the use of special eviction sets and then applies to it common side-channel attacks.

Just this year, Lipp et al. [19] discovered the Collide+Probe and Load+Reload attacks which target AMD processors' caches. These attacks efficiently leak sensitive data from the cache by specifically targeting the processors' cache way predictors.

As we can see, cache timing vulnerabilities are ever increasing in number and come in a multitude of forms. However, such vulnerabilities can be used in other security exploits as well.

2.2 Cache Attack Applications

The vulnerabilities above have been extensively used to augment other hardware security exploits. In this section, we describe several exploits that utilize cache attacks to very effectively leak sensitive data.

2.2.1 Cache-based Exploits

Cache Template Attacks [9] is a strategy that automates cache-based security vulnerability discovery and exploitation. In this strategy, the attack breaks up an application's execution into events. The attack then uses Flush+Reload to associate each event with a set of memory and cache access patterns. During subsequent application runs, the attacker utilizes Flush+Reload in combination with the event-to-access pattern association to infer the occurrence of events from the application's accesses.

2.2.2 Out-of-Order and Speculative Execution Exploits

The applications of cache attacks are not just limited to other cache-based exploits. Meltdown [18] is a simple and generalizable out-of-order execution-based attack that can leak sensitive information. In this exploit, the attacker triggers an exception by accessing sensitive data. This data is then loaded into a pseudo-register for out-of-order instructions to use. These instructions access an array, with the index depending on the secret data. During out-of-order execution, the cache is able to be modified, and thus the accessed array data is stored in a cache line whose location is based on the sensitive index. Cache side-channels such as Flush+Reload are then used to find the cache line that was loaded in and thus reveal the sensitive index to the attacker.

Spectre attacks [15] leverage speculative execution, such as branch prediction, to leak victim data. In one variant of a Spectre attack, the attacker induces mispredictions on conditional branches that jump to data leaking instructions. In another Spectre variant, the attacker causes indirect branches to jump to specific code in the victim's address space called "gadgets". These gadgets are then speculatively executed to load sensitive data into the cache. In both variants of Spectre, the covert channel is similar to Meltdown's: they both use cache attacks to leverage the fact that changes to the cache (or other microarchitectural states) are made during out-of-order or speculative execution.

Thus, as there are so many cache timing vulnerabilities and ways to exploit them, it is imperative that designs be debugged of all timing-related security bugs.

2.3 Hardware-Based Cache Side-Channel Mitigations

In this section, we give some examples of hardware-based cache attack mitigations. Most hardware-based cache side-channel mitigations fall under three strategies: Complete Cache Access Avoidance, Eviction Set Creation Prevention, and Victim Cache Access Inference Prevention.

2.3.1 Complete Cache Access Avoidance

A guaranteed way to mitigate cache attacks is simply not to use the cache at all. For example, AES-NI [10] is a set of x86 instructions that allow for hardware-based AES encryption. Not only do these instructions avoid cache accesses, but they also execute in constant time, making it impossible for any timing attack to leak secret information. Mitigations that avoid the cache, however, are impractical as they are either inefficient or, like AES-NI, have to be tailored towards specific applications.

2.3.2 Eviction Set Creation Prevention

As discussed, conflict-based cache attacks rely on eviction sets to cause conflicts with victim lines. Preventing eviction set creation effectively stops these attacks. One method of preventing eviction set creation is the randomization of physical address to cache location mappings. NewCache [21] and the Random Permutation Cache [34] achieve this randomization through table-based indirection, with the former utilizing content addressable memory and the latter relying on, as the researchers call it, a custom “permutation table”. CEASER [27] on the other hand randomizes the mappings through the encryption of physical addresses before cache accesses are made.

2.3.3 Victim Cache Access Inference Prevention

These types of mitigations are the most common and possess a wide variety of strategies ranging from randomizing timers [22] to locking sensitive lines in the cache [34].

An effective method for protecting a victim’s cache accesses involves partitioning the cache into secure regions controlled by specific processes. The PLCache [34] facilitates such a partitioning by allowing processes to dynamically lock and unlock their own data in the cache. When a process locks a sensitive cache line, other processes cannot evict it through cache conflicts. Attacks like Percival’s Attack and Prime+Probe are therefore thwarted, as in their reload/probe steps, they will always observe that the victim had accessed the sensitive cache line.

As discussed in the previous section, many cache attacks require inclusive caches to properly work. The Relaxed Inclusion Cache [14] observes this requirement and thus eliminates cache inclusiveness for critical data used by cryptographic algorithms. By making such data non-inclusive, those inclusivity reliant attacks now fail.

Since a multitude of cache attacks rely on timing cache reloads or accesses, preventing the attacker from precisely timing such latencies is another mitigation strategy. TimeWarp [22] achieves this by introducing randomness to timing instructions (i.e. `RDTSC`).

Other mitigation techniques include the Secure Hierarchy-Aware Cache Replacement Policy (SHARP) and the Prefetch-Obfuscator to Defend Against Cache Timing Channels (PrODACT) [36, 7]. SHARP [36] modifies the LLC cache replacement policy such that "inclusion victim" creation is reduced. The mitigation defines inclusion victims as addresses that are evicted from a victim process's L1 cache when they are evicted from the LLC. By reducing the number of inclusion victims, victim data will often reside in the L1 cache after an eviction, in turn confusing the probe/reload steps of cache attacks. PrODACT [7], on the other hand, continuously checks the LLC, specifically its recent conflicts, for any ongoing attacks. When it detects an attack, it utilizes the prefetcher to hide and thus protect victim processes' accesses.

In order for the design and implementation of timing secure caches and cache attack mitigations to be practical, tools for automatic security verification and debugging are a necessity. Without such tools, the designer is prone to making errors and wasting resources during the design and implementation phases. For example, we know that the PLCache contains a previously unknown timing-related side-channel [3]. Even if the designer can guarantee that their overall design is secure, they still need these tools to verify that the design's actual implementation in Hardware Description Languages (HDLs) like Verilog is secure. Therefore, in order to develop secure caches, it is extremely important for the designer to exhaustively verify and debug the implemented mitigations through the use of automated tools.

2.4 Other Insecure Hardware

While caches are a major source of timing-related security bugs, they are not the only hardware components that are timing insecure. Ardeshiricham et al. [1] have shown that an

RSA core leaks data through modular exponentiation-related timing variations. They have also demonstrated that shared bus architectures such as WISHBONE allow for the leakage of sensitive information among connected hardware components through timing variations. In general, any hardware core that exhibits timing variations based on their inputs or operations may leak sensitive data. Therefore, all hardware that suffers from timing variations, not just caches, should be verified for timing flow security. Current formal hardware verification strategies such as Information Flow Tracking (IFT) are able to do just that.

2.5 Information Flow Tracking

Hardware Information Flow Tracking (IFT) is a general pre-silicon security verification technique that assigns sensitivity labels to data in a hardware design. Labels that indicate sensitive data usually assume a high value and are called tainted. Those that have low values are called untainted. As the data propagates, or flows, throughout the system during a simulation, the labels' values are also propagated depending on the operations involved (flow tracking). This label propagation allows for the verification of security properties. For example, one might want to verify that sensitive information, such as a secret key, has not leaked into some output data by asserting that the output data's label is not tainted. In this section, we discuss and detail current works in hardware IFT and works that utilize IFT to ensure the creation of secure designs.

2.5.1 Timing vs. Functional Flows

Oberg et al. [23] identified two types of information flows: functional and timing. According to them, "a functional flow exists for a given set of inputs to a system if their values affects the values output by the system (for example, changing the value of a will affect the output of the function f(a, b): = a + b)" [23, p. 5]. Note that the "system" they refer to doesn't need to be a function or module. Functional flows can occur in computations as simple as one variable propagating its value to another. Meanwhile, a “timing flow exists if changes in the input affect how long the computation takes to execute” [23, p. 5]. For example, a simple counter function that executes for its input value of cycles possesses a timing flow. In this paper, we adopt Oberg et al.’s information flow terminology. As we are interested in sensitive data leakage through timing variations, we will primarily focus on timing flows in this paper. We now provide several examples of information flow tracking techniques, including one whose approach we will utilize in this paper.

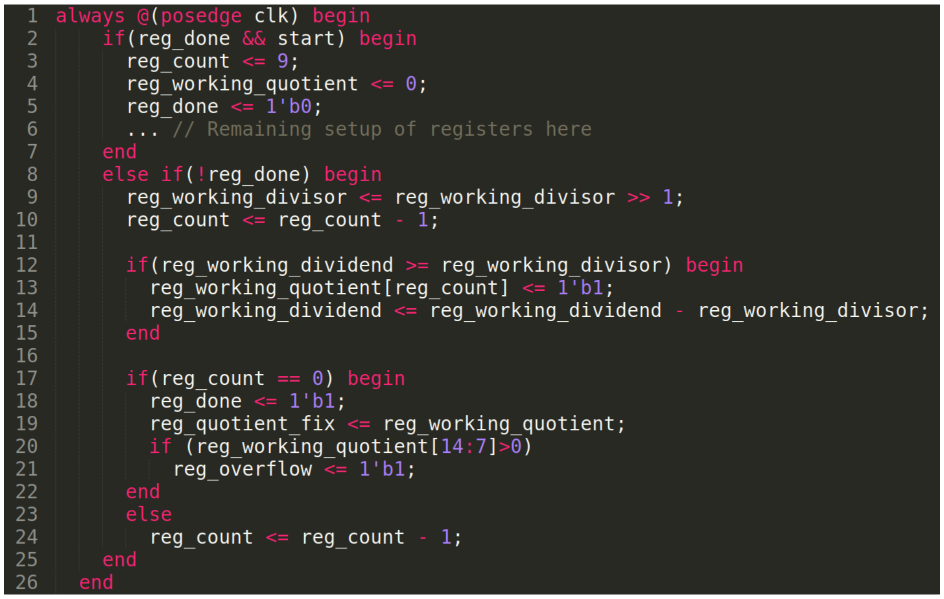
2.5.2 Single Label IFT Techniques

Gate Level Information Flow Tracking (GLIFT) [32] is a technique that tracks information flow through low level logic gates. This technique assigns security labels with value 1 to indicate data containing sensitive information and value 0 to indicate data containing non-sensitive information. For every gate, GLIFT implements shadow logic that uses both the incoming data and labels to infer the label value of the output. These shadow logic units can be composed to form any larger shadow functional unit, meaning that a wide range of hardware designs can be monitored for security with GLIFT. Moreover, GLIFT can track both timing and functional flows [23, 32].

Register Transfer Level Information Flow Tracking (RTLIFT) [2] possesses the same data flow tracking idea as GLIFT, but tracks data at the higher Register Transfer Language (RTL) Level. RTLs model the relationships and signal flows between registers in hardware designs. The authors of RTLIFT evaluated this technique and found that it achieves less complexity, faster verification, greater flexibility, and fewer false positives compared to GLIFT. As with GLIFT, RTLIFT is able to track both timing and functional flows.

2.5.3 Multiple Label IFT Techniques

Ardeshiricham et al. [1] made the observation and analysis that while GLIFT, RTLIFT, and other techniques can track all forms of information flows, both timing and functional,



```verilog
always @(posedge clk) begin
    if(reg_done && start) begin
      reg_count <= 9;
      reg_working_quotient <= 0;
      reg_done <= 1'b0;
      ... // Remaining setup of registers here
    end
    else if(!reg_done) begin
      reg_working_divisor <= reg_working_divisor >> 1;
      reg_count <= reg_count - 1;

      if(reg_working_dividend >= reg_working_divisor) begin
        reg_working_quotient[reg_count] <= 1'b1;
        reg_working_dividend <= reg_working_dividend - reg_working_divisor;
      end

      if(reg_count == 0) begin
        reg_done <= 1'b1;
        reg_quotient_fix <= reg_working_quotient;
        if (reg_working_quotient[14:7]>0)
          reg_overflow <= 1'b1;
      end
      else
        reg_count <= reg_count - 1;
    end
  end
```

Figure 2.1: Procedural block of a divider module that generates and blocks sensitive timing flows [5]. This example is similar to the pseudo code example found in Figure 1 of the Clepsydra paper [1]. Our analysis of this example is also based on the Clepsydra paper's explanation of its own example.

they usually employ only one type of label, which is not enough to differentiate between flows that are caused by timing (e.g. timing variations) and functional insecurity (e.g. secret key bytes affecting cipher text). The researchers explained that if we only wanted to track timing flows, a number of problems like false positives would arise. Ardeshiricham et al. also discussed a different problem in which the output of a cryptographic core would always have a high label value since it is always affected by the sensitive key. They would not specifically know if the core has timing variations that depend on the input.

Clepsydra [1] and VeriSketch [3] define an IFT approach that solves the problems above by separating information flow labels into functional and timing (thus, all data has two labels). The papers defined and then proved specific rules that account for the generation, propagation, and blocking of tainted timing flows. Clepsydra and VeriSketch determined that tainted timing flows are generated by unbalanced conditional assignments to registers that are within procedural blocks with sequential logic (sequential blocks) and controlled by signals with tainted functional flow. According to Ardeshiricham et al, an unbalanced assignment, or "unbalanced update means that there exists a clock cycle where register v can either maintain its current value or get reassigned" [3, p. 6]. Moreover, the researchers determined that tainted timing flows propagate to a register if its assigned value or one of its assignment's control signals possesses tainted timing flow. Note that a control signal is one that plays a role in deciding which value should be assigned to a register at clock edges. The researchers also determined that tainted timing flows are blocked at a register within a sequential block if at least one of its control signals is non-sensitive (only has untainted flows) and fully controlling. According to Ardeshiricham et al., a control signal for a register is fully controlling when "the register gets a new value if and only if the controller gets a new value" [1, p. 3].

To illustrate how tainted timing flow generation and blockage works, we examine Figure 2.1, which shows the procedural block of a timing secure division module that was implemented by Burke and can be found on OpenCores [5]. This example is similar to the example corresponding to Figure 1 in the Clepsydra paper [1, pg. 3]. Our analysis of the example in Figure 2.1 is also based on the Clepsydra paper's explanation of its own example. Note, however, that the example we show is represented with real Verilog code. Assume that *reg_working_dividend* initially possesses a tainted functional label and an untainted timing label. Also assume that all other registers' labels are untainted. Note that the register $reg\_quotient\_fix$ is the output register for the module. The registers *reg_working_quotient* and *reg_working_dividend* are unbalanced since at every cycle, they can either be updated or remain the same (unbalanced update at lines 13 and 14). Moreover, *reg_working_dividend*, which has tainted functional flow, is one of *reg_working_quotient*'s and *reg_working_dividend*'s control signals at lines 13 and 14 respectively. Thus, tainted timing flow is generated at *reg_working_quotient* and *reg_working_dividend* when $reg\_working\_dividend \geq reg\_working\_divisor$. Conceptually, the two registers having tainted timing flows make sense because the time at which their output values update depends on the value of sensitive data. Even though at line 19 $reg\_quotient\_fix$ is assigned *reg_working_* *quotient*'s value, the tainted timing flow from the latter register is blocked from propagating to the former register. The timing flow from *reg_working_quotient* to *reg_quotient_fix* is blocked because *reg_done* is a non-sensitive and fully controlling signal of *reg_quotient_fix*'s updates. *reg_done* is non-sensitive because it lacks tainted functional and timing flows. The controller is also fully controlling as it only updates when *reg_quotient_fix* is updated: when the counter, *reg_count*, becomes 0. This timing blockage example conceptually makes sense because the output is always written at a constant time, meaning that there are never any timing variations.

IFT is a powerful tool for automatically detecting sensitive data leakage in hardware designs. However, as IFT is primarily a hardware verification tool, it does not, in its current form, find the origin of such bugs. If the programmer wishes to fix the bug, they must manually analyze the design through methods such as waveform analysis and check pointing. These methods are often inefficient and time consuming, especially when the hardware design is large. Tools that can efficiently and automatically find a bug's location are therefore sorely needed.

2.5.4 Using IFT to Ensure Secure Hardware Designs

If the introduction of security bugs can be eliminated entirely during the construction of a design, then debugging and thus bug localization would not be required. In addition to its IFT approach, VeriSketch [3] is also an automated hardware verification and synthesis technique. That is, this technique is able to automatically determine whether a design written in VeriSketch's sketch syntax conforms to some specifications (e.g. security or behavioral). If the design fails verification, VeriSketch modifies and then verifies the design continuously until either all properties are ensured or a proper design cannot be found. Note that it was through the VeriSketch's analysis on the PLCache that we know that the mitigation has a timing-related vulnerability. After they detected a security bug in the PLCache design, VeriSketch's creators manually found a side-channel in the mitigation that is induced by its uniform LRU policy for all cache lines. The researchers took advantage of this behavior and were able to induce timing-based leakage. While VeriSketch is an effective synthesis tool, it lacks the scalability to large and complex designs. For example, VeriSketch's researchers found that it took around six to eight hours to synthesize a secure cache design. The application of VeriSketch on any larger and/or more complicated designs (e.g. an entire processor) could therefore require hours or days. In addition, while VeriSketch allows the programmer to customize the performance and behavior of a synthesized design through user-provided soft constraints, the user does not possess full control over the specific structure and behavior of the design. The designer must also learn to write designs in VeriSketch's sketch syntax. Finally, there is a chance that VeriSketch fails to synthesize a design that conforms to the specifications, requiring the designer to perform more work to either modify the design sketch or specifications. All of these problems therefore lead us to believe that widespread use of VeriSketch is unlikely.

Researchers have also extensively explored the idea of specially designed HDLs that employ IFT approaches to prevent the designer from ever violating some defined security properties. The HDLs SecVerilog [39] and Caisson [16] implement IFT-based typing systems that are used to verify a hardware design's security. Sapper [17] is an HDL that enforces a design's security by giving the compiler the ability to add to Verilog code IFT-related assertion logic. A wide adoption of such HDLs would greatly reduce the number of security bugs and thus the need for automatic bug localization. However, programmers would have to learn how IFT works, how to code in different languages than common HDLs, and the more restrictive design patterns enforced by these languages [1, 2]. Moreover, each these languages suffers from some overhead (e.g. area, power, performance, manual effort) in order to impose IFT rules and security [2, 16, 17, 39]. We therefore currently do not foresee the widespread adoption of these new HDLs.

VeriSketch's and the HDLs' problems and potential lack of adoption lead us to believe that, for the foreseeable future, it will be up to the designer to manually find and fix security bugs. As these bugs are often subtle and difficult to find, tools that can help to localize a bug in a hardware design are a necessity.

2.5.5 Bug Localization

One of the most naive methods for localizing timing-related security bugs is manual debugging through the use of wave forms, checkpointing, replaying, code analysis, incremental code/variable modifications, etc. However, manual localization methods are often tedious and inefficient, especially for large designs, as the programmer likely has to analyze thousands of signals and an exponential amount of data paths over a multitude of cycles. Moreover, the designer has to know the design's structure very well, which may not always be true. Automated and efficient techniques are therefore a necessity to help the designer construct secure designs. We now discuss some more automated localization techniques.

Automatic location and root cause detection has been explored and demonstrated in the post-silicon debugging world. IFRA [25] is an instruction trace-based debugging strategy for CPUs. It utilizes recorders to capture information about instructions at the processor's pipeline stages. During the hardware testing phase, applications or synthetic traces are run on the CPU. IFRA stops the CPU when some errors are detected (e.g. segfault) and outputs the recent instruction history stored in each recorder (i.e. few thousand instructions). The technique then combines all of the histories and then analyzes them against four main heuristics. In doing this, IFRA is able to localize a bug with high accuracy. However, because IFRA relies on four set heuristics to localize a bug, the types of bugs it can localize are limited.

Coppelia [40] is a post-silicon verification technique that utilizes a form of backward symbolic execution that traverses execution flow trees backwards from a buggy state to programatically create CPU exploits. The programs that Coppelia generates are often extremely small, which means that determining the flaw's location is made easier. However, the designer must still manually analyze the exploit program to determine the location of the bug (e.g. through replaying or eye level analysis). Moreover, Coppelia can only find and generate functional exploits since its symbolic execution must start at an invalid architectural state; timing exploits are not related to invalid states.

Thus, while there have been works in or methods for localizing bugs in hardware, those works/methods are limited in scope, lack full automation, are inefficient, and/or, most importantly, cannot localize timing-related security bugs. To the best of our knowledge, few, if any, works have specifically explored automated timing-related security bug localization in hardware designs. Moreover, we believe that this is the first work that uses IFT to localize timing-related security bugs. We now describe in detail our main research problem and goals.

2.6 Research Problem and Goals

As we have seen, a multitude of hardware designs such as caches, cache-attack mitigations, and cryptographic cores may suffer from timing-based security vulnerabilties. We can utilize IFT strategies to verify that hardware-based mitigations satisfy side-channel mitigation invariants. However, localization of a detected timing bug must still be done manually. For example, there is no automatic localization procedure in VeriSketch. Moreover, current solutions that ensure secure designs lack the flexibility and simplicity for widespread adoption. Therefore, we seek to solve the following problem:

- Assuming that there exists timing-based leakage in a hardware design, how can we automate the localization for that bug?

Our main goal is thus:

- Develop an automated simulation-based strategy that utilizes VeriSketch's/Clepsydra's IFT approach in combination with faulty simulation traces to accurately localize a detected timing-related security bug in a hardware design.

We provide the following contributions in this paper:

1. We present seven automated simulation-based techniques that localize security bugs by analyzing IFT behaviors during faulty simulations. We call these techniques:
    - Selective Taint Generation
    - Taint Generation Logging
    - Brute-Force Backtracking
    - The Aggregated Graph Walking Technique
    - Unique Taint Tracking
    - The Bloom Filter Approach
    - Probabilistic Pathing
2. We evaluate the techniques' accuracies through various methods that include empirical experiments on example hardware designs, qualitative exploration on handmade/theoretical examples, and informal reasoning. We also evaluate the techniques' efficiencies with informal worst-case computational complexity analyses.
    - Note: the worst-case computational complexity analyses are informal and not perfectly precise. However, we believe that the analyses are enough to provide a general idea of the techniques' efficiencies.
3. We discuss and summarize the results from the techniques' evaluation. Moreover, we construct tables that neatly compare the techniques.

CHAPTER 3
DEFINITIONS AND ASSUMPTIONS

We construct seven techniques that localize security flaws in Verilog RTL hardware designs by analyzing the IFT signals and behaviors of security violating (faulty) simulations. In this section, we describe our definitions and assumptions for these techniques. Note that terms we define here will apply in subsequent sections of this paper.

3.1 Assumptions

We discuss our techniques under the following assumptions:

1. We assume that the designer has already instrumented the debugged design with Clepysdra's/VeriSketch's IFT rules for security verification.
2. We assume that the design contains no bugs, both security and functional, other than timing-related security bugs.
3. The inputs to the design do not contain tainted timing flow that affect the outputs' timing flow labels. We assume this as we are primarily interested in how bugs are generated by the hardware itself.
4. The design possesses only one security bug (see definition below).

3.2 Definitions

The definitions we provide in this section form the basis of discussion related to our techniques.

- **Module:** Modules in a Verilog RTL design.
- **Signal:** A wire or register in a Verilog RTL design. This definition also encompasses IFT wires or registers corresponding to security labels.

- **Monitored Labels:** The set of timing flow security labels that form the timing-related security invariants of a hardware design. If during a simulation at least one label possesses an erroneous value (i.e. high, non-zero), then the design possesses a timing-related security bug. These labels are always monitored during a simulation under IFT.

- **Timing-Related Security Bug:** A timing-related security bug occurs when there is a violation of the monitored labels/security invariants.

- **Register Adjacency:** Two registers r_1 and r_2 are adjacent if one's timing flow label explicitly or implicitly depends on the other's through combinational logic. That is, there exists a path between r_1 and r_2 that contains no registers other than r_1 and r_2.

- **Register-Wire Adjacency:** A wire w and register r are adjacent if one's timing flow label explicitly or implicitly depends on the other's through combinational logic. That is, there exists a path between w and r that contains no registers other than r.

- **Insecure Path:** A continuous path of adjacent, tainted timing flow carrying, registers that begins somewhere in the design and ends on a register that either possesses a monitored label, or is adjacent to and affects at least one of the monitored signals. That is, these paths are register paths on which the tainted timing flows propagate without blockage to the monitored label(s). Note:

  - Every insecure path contains the taint generating source.
  - A bug may have multiple insecure paths as the tainted timing flow can be generated at more than one register.
  - These insecure paths may overlap because registers can both propagate and generate flow at the same time.
  - By definition, any register along an insecure path affects a monitored label.

- **Location of a Timing-Related Security Bug**: The location of a timing flow-related security bug is comprised of the registers whose conditional assignments generate tainted timing flows whose values, in turn, affect the values of the monitored labels. A timing bug location, from another perspective, consists of registers that generate tainted timing flows which in turn propagate to the monitored labels. Thus, when we say bug location, we mean the set of registers where security invariant/monitored label affecting tainted timing flows are generated. It is important to note that the insecure path(s) are equally as important because it helps the programmer know where to block the timing flow. Moreover, under Clepsydra's and VeriSketch's IFT rules, tainted timing flows can only be generated at procedural assignment to registers [1, 3].

- **Faulty Simulations:** Faulty simulations are those whose inputs cause the monitored/asserted on IFT label security invariants to be violated.

- **Data Loops:** A register x is in a data loop with another register y if x affects y and y affects x.

- **Disjointed Register:** A register is disjointed if the specific tainted flows it carries never touch any monitored labels. That is, at some point, the taints are stopped from propagating to the monitored labels (either by input choices at multiplexers or timing flow blockage points).

CHAPTER 4
SELECTIVE TAINT GENERATION

One of the most straightforward ways to localize a timing-related security bug is to test each unbalanced register r for bug location membership by: 1) turning off the tainted timing flow generation logic for all registers except for r and then 2) running the simulations with the modified HDL and checking if the monitored labels are tainted. This procedure is possible as 1) tainted timing flows can only be generated at unbalanced registers and 2) the timing flow generation and propagation logic are independent of each other. We now describe this procedure in detail.

4.1 Detailed Description

Based on Clepsydra's [1] timing flow logic descriptions and rules, we have the following logic for timing flow propagation and taint generation for a procedural assignment $r <= r'$ that resides within a sequential block:

$$r_{time} <= (r'_{time} \ \& \ !(ns\_full(s_0) \mid ns\_full(s_1) \mid ... \mid ns\_full(s_n))) \mid \tag{4.1}$$

$$(s0_{time} \mid s1_{time} \mid ... \mid s_{n_{time}}) \mid \tag{4.2}$$

$$((s0_f \mid s1_f \mid ... \mid s_{n_f}) \ \& \ !is\_bal(r)) \tag{4.3}$$

Lines 4.1 and 4.2 constitute the timing flow propagation logic and line 4.3 comprises the taint generation logic. Any variable subscripted with $time$ is a timing label and any variable subscripted with f is a functional label. The set $\{s_0, s_1, ..., s_n\}$ is the set of control signals for $r <= r'$. The function $ns\_full(s_i)$ determines whether s_i is a non-sensitive fully controlling signal of r, and $is\_bal(r)$ determines whether the assignments to r are balanced assignments. Therefore, the generation and propagation portions of the timing flow logic are independent of each other and can independently be turned off and on.

Now, given a set of simulations, this technique checks the designs' registers for timing generation through an iterative process. It first identifies all registers with unbalanced updates (call this set of registers U). A subset of U will be the bug location, call it U_S. In order to determine the exact subset, the technique does the following: let U_S be empty. For each register $r \in U$,

1. The technique disables all registers' tainted timing flow generation logic except for r's. We leave the timing flow propagation logic enabled for every register. More specifically, for every register other than r, we disable its line 4.3 logic while leaving lines 4.1 and 4.2 enabled. Line 4.3 can be disabled through code deletion or comments. r's timing flow code is left untouched.

2. The technique then compiles and runs the faulty simulations again. If the monitored labels are tainted in any simulation, then we know that the register is part of the bug location. r is therefore added to U_S.

After all unbalanced registers have been tested, the registers in U_S are returned as the bug location.

4.2 Accuracy Evaluation

This technique clearly always achieves 100% accuracy. Let the register r generate tainted flow but also be disjointed. When this technique tests r, it will always observe this disjointed behavior because only the taint generation logic is disabled; the logic handling flow propagation/blockage is not touched. The technique will thus accurately recognize r as a non-buggy register. On the other hand, if r generates tainted flow that propagates to the monitored labels, the technique will observe this behavior and correctly recognize r as a bug location register. We therefore do not conduct an in-depth accuracy evaluation for this technique.

4.3 Efficiency Evaluation

In this section, we informally reason about this technique's worst-case time and space complexities. Note that this analysis is not perfectly precise (nor do we claim that it is perfect), but it does a good job of providing a general idea of the technique's efficiency. We define the following variables for our complexity analysis:

- s = number of simulations.
- u = number of unbalanced registers that affect the monitored labels. Note that the number of taint generating registers is less than or equal to u.
- m = number of monitored labels.
- n = the maximum number of operations in an unmodified simulation.

We define the following costs for our complexity analysis:

- $O(1)$ = the worst-case time cost of executing one of a simulation's operations at a cycle (e.g. IFT logic, register assignments, and combinational logic assignments). Note then, that the worst-case time complexity of a simulation is $O(n)$ and the time complexity of running all simulations is $O(sn)$.
- $O(1)$ = the time complexity of checking whether the monitored labels were violated during a simulation.
- $O(x)$ = the maximum memory footprint of an unmodified simulation.
- $O(k)$ = the maximum time it takes to modify and recompile the design. We assume that $k >> 1$.
- $O(1)$ = the memory footprint of a monitored label during the localization part of the technique.

Furthermore, for simplicity, we assume that the action of logging the simulations' behavior (registers, IFT, control signals) takes $O(1)$ time and space.

4.3.1 Time Complexity

In order to reason about the time complexity, we split the technique into two parts: simulation and localization. We first reason about the complexity of the simulation part. This technique needs to modify and recompile the design for each unbalanced register. Moreover, for each recompile, we need to run all the simulations on the recompiled design. The time complexity for the simulation part is therefore, in the worst-case, $O(u(k + sn))$. The localization part of the technique needs to check whether the simulations for each unbalanced register are faulty or not. Therefore, in the worst-case, the complexity is $O(us)$. Thus, the overall worst-case complexity is $O(u(k + sn)) + O(us) = O(u(k + sn))$

4.3.2 Space Complexity

Like above, in order to calculate the space complexity, we split the complexity into two parts: simulation and localization. This technique adds no additional memory overheads during simulation and thus the complexity is $O(x)$. Note that $O(sx)$ space is not required because each simulations' memory can be freed after it finishes. For the localization step, the technique needs to keep the monitored labels' final values. This results in an $O(sm)$ space complexity. Note that $O(usm)$ space is not required because the monitored labels' memory can be freed after each register test. Thus, the overall worst-case complexity is $O(x + sm)$.

CHAPTER 5
TAINT GENERATION LOGGING TECHNIQUE

The Taint Generation Logging Technique determines the set of registers that generate tainted timing flow during faulty simulations and proposes it as the bug location. This set of registers at the very least contains all the registers that are part of the bug location. This strategy consists of two phases, Pre-Processing and Simulation and Candidate Identification, which we will now describe in detail.

5.1 Detailed Description

In order to discuss the technique, we provide the following definitions.

Definitions:

- Let I_f be the set of faulty inputs.
- Let R be the set of registers monitored in the design. The initial state is empty.
- Let R^* be the set registers in R that carry tainted flow in a faulty simulation. The initial state is empty.
- Let R' be the set of registers in R^* that generate tainted flow in a faulty simulation. The initial state is empty.
- Let U be the set of unbalanced registers that are also in R.
- For $i \in I_f$, let c_i be the number of cycles in the simulation that takes in i as input.
- For $i \in I_f$, let T_i be the signal trace matrix for all $r \in R$ during the simulation on i. It will be of size $|R| \times c_i$.
- For $i \in I_f$, let G_i be the taint generation trace matrix for all $u \in U$ during the simulation on i. It will be of size $|U| \times c_i$.

5.1.1 Pre-Processing

During the Pre-Processing phase, the technique identifies the registers whose timing labels indirectly or directly affect the monitored labels. It places them in R. The technique then filters R down to the set of registers U that have unbalanced updates. Note that this unbalanced register identification process is feasible as both Clepsydra and VeriSketch statically analyze the hardware design's AST to identify unbalanced updates to registers [1, 3]. Since we want to find the registers that generate tainted timing flow, for each register $u \in U$, the technique adds to the design a register called $u\_is\_generation$, which tracks whether tainted timing flow is ever generated at the register u. Now consider the timing flow propagation and taint generation logic we described in Chapter 4 (lines 4.1 to 4.3). The following code logic is added to the design for each of u's unbalanced conditional assignments $u <= x$:

$$u\_is\_generation <= s_{0,f} \mid s_{1,f} \mid ... \mid s_{n,f};$$

The right hand side of this logic is exactly the tainted timing flow generation logic but without the unbalanced assignment checking logic. We do not include the unbalanced assignment checking logic because we have already identified $u <= x$ as unbalanced.

Output: A design augmented with taint generation tracking code and the register set R.

5.1.2 Simulation and Candidate Identification

In this phase, the technique simulates the design on a multitude of faulty inputs. For each simulation, the technique keeps trace matrices that keep track of registers' *is_generation* signal and timing label values. The technique then analyzes these traces to identify candidate bug location registers. We now describe each step of this phase in detail.

During the Simulation and Collection step, for each simulation of the augmented design on an input $i_f \in I_f$, we record the high-low values of r's timing label for all $r \in R$. We also record u's *is_generation* label for all $u \in U$. Rather than terminate the simulation the moment the security invariants are violated, we let the faulty simulation run to completion in order to capture all taint generating registers and insecure paths. Terminating early may leave out some registers that later generate tainted flow that in turn propagates to the monitored labels. The result for each simulation consists of the two matrices T_{i_f} and G_{i_f}. Each row of T corresponds to a register's timing label. Each column corresponds to a cycle of the simulation i_f. If a register's timing label possesses a non-zero (i.e. high) value for a specific cycle, then the corresponding $T_{i_f}[register, cycle]$ entry in the matrix is given the value 1. Otherwise, the value in that entry is set to 0. Similarly, each row of G corresponds to an unbalanced register's *is_generation* label and each column corresponds to a cycle in the simulation. If a register's *is_generation* label has a high value at a specific cycle, then $G_{i_f}[register, cycle]$ is set to 1. Otherwise, the technique sets the entry's value to 0. For example, let the simulation on i_0 run for 4 cycles. Moreover, let $|R| = 2$ and $|U| = 2$. Then, we may have the two trace matrices (1 denotes high label value and 0 denotes low):

$$T_{i_0} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \text{ and } G_{i_0} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

The first row of T_{i_0} corresponds to one register's timing label and the second row corresponds to another's. This first register's timing label, in the first cycle, is tainted. Then the label assumes a low, low, and finally high value. The second register, on the other hand, always carries a tainted timing flow. The first row in G_{i_0} corresponds to the first register's *is_generation* label. Since the row is all 0s, the first register never generates tainted flow. The second row's register's *is_generation* label, on the other hand, has a value of 1 for all cycles and thus generates tainted timing flow at all cycles.

Once the previous step finishes, the technique begins the Candidate Identification step by considering all the traces that are derived from simulations on faulty inputs. That is,

$$\{T_{i_f} \mid i_f \in I_f\} \text{ and } \{G_{i_f} \mid i_f \in I_f\}$$

For each register in R, the technique searches the corresponding rows in each T_{i_f} until it finds an entry with the value 1. Once it finds an entry with value 1, the register must have carried tainted timing flow and thus the technique adds the register to the set R^*. If the technique never finds an entry with value 1, it does not add the register to R^*.

Similarly, for each register in U, the technique in this step searches the corresponding rows in each G_{i_f} until it finds an entry with the value 1. Once it finds an entry with value 1, the register must have generated tainted timing flow and thus the technique adds the register to the candidate set R'. If the technique never finds an entry with value 1, it does not add the register to R'. At this point, the Taint Generation Logging Technique proposes R' as the candidate set and also provides the programmer with R^* for the security bug's context. This technique then terminates.

Output: The candidate set R' and the set of taint carrying registers R^*.

5.2 Experimental Methodology

We evaluated this technique's accuracy both through empirical experiments and qualitative exploration on handmade and theoretical examples. Before we report our evaluation and results, we first describe the examples from the empirical experiments and how we ran those experiments. In this study, we tested the technique's bug localization ability on the following buggy examples instrumented with IFT logic: a multiplier, a divider, AES cores injected with faulty dividers, an RSA core, and a direct mapped cache. The IFT instrumented Verilog code for these examples was generously provided by the Kastner Research Group, whose members (current and former) are the authors of papers such as [1, 2, 3, 17, 23]. Unless specified, the examples do not possess any registers that block timing flows.

5.2.1 Validating Candidate Set

Once we had implemented this technique and were ready to test it on our examples, we needed a procedure that allowed us to check whether the technique proposed the correct registers. We therefore utilized the technique described in Chapter 4 on the examples in order to verify the correctness of the Taint Generation Logging Technique's suggested bug locations.

5.2.2 Multiplier

We experimented on a multiplier that exhibits timing variations when the multiplication of the inputs causes an overflow. Note that the multiplier's original, non-IFT, code was implemented by Burke and can be found on OpenCores [5]. Using the validation procedure in Chapter 4, we found that bug resides at the registers *qmults.reg_overflow* and *qmults.reg_working_result*. The design contained 8 unbalanced registers. In this example, we utilized 136 random faulty inputs. In all inputs, the multiplicand and multiplier signals contained sensitive data and thus had a tainted functional label. All of the multiplier's output labels were monitored, i.e., *result_seq_time*, *complete_time*, and *overflow_time*.

5.2.3 Divider

The divider design is a subtraction-based division core whose outputs exhibit timing variations depending on the dividend's value. As with the multiplier design, the original code was implemented by Burke and can be found on OpenCores [5]. We found with Chapter 4's procedure that the exact buggy registers are *qdiv_orig_ift.reg_working_quotient*, *qdiv_orig_ift.reg_working_dividend*, and *qdiv_orig_ift.reg_overflow*. The design overall contained 7 unbalanced registers. In this example, we simulated on 181 random faulty inputs. All dividend and divisor inputs carried tainted functional flow. That is, they contained sensitive data. All of the divider's output labels were monitored, i.e., *quotient_out_orig_ift_time*,

$complete\_orig\_ift\_time$, and $overflow\_orig\_ift\_time$.

5.2.4 AES Cores Injected with Divider

These examples consist of AES cores injected with the divider module described above. Note that the original, non-IFT, AES core is part of the benchmarks used by Salmani et al. [29] and Shakya et al. [30] and can be found on Trust-Hub [28]. While the original AES core does not possess any timing-related bugs, the injected divider module synthetically creates one. These examples include: a base example, an example with large data loops, and finally an example with timing flow blockage points. The base example is almost the same as the original AES core, but with an $expand\_key\_128$ module called $a5$ injected with the previous example's divider core. The divider core divides the original output of $a5$ by a random constant and pushes its own output to $a5$'s output. $a5$ is located at around the middle of the AES design. Because the only buggy module in this design is the divider core, the buggy registers are the same as in the previous example, but without the overflow register: $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_dividend$ and $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_quotient$. The overflow register is excluded because only the quotient is used by $a5$, not the overflow value. That is, the overflow register does not affect the monitored labels. In this example, we experimented with 85 random faulty inputs.

The loop example possesses a data loop between an $expand\_key\_128$ module, called $a2$, and the buggy module $a5$, whose location is the same as in the base example. More specifically, $a5$'s (possibly tainted) outputs, when available, are sent backwards in the design as $a2$'s inputs. We call this case "Loop". In this case, we added no additional registers and the buggy registers are $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_dividend$ and $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_quotient$. We utilized 87 random faulty inputs for this case.

The final example was the base AES with divider example injected with an additional timing secure divider module. This timing secure divider always blocks any generated timing flow from reaching its output register. We placed the secure divider module within an *expand_key_128* module called *a*2 that the *a*5 module, which the faulty divider resides in, depends on. The faulty divider module's inputs thus depend on the secure divider module's output quotient. Because the secure divider module blocks all tainted timing flows, the bug location in this modified example is exactly the same as the AES with divider base example's. We simulated the design on 87 faulty inputs.

In the first two examples, 6 registers were unbalanced. In the third example, 12 registers were unbalanced. Moreover, each key and state input carried sensitive data. We also monitored the AES core's output's label (*state_out_time*) for all three examples.

5.2.5 RSA Core

This RSA core possesses a timing bug in the modular exponentiation step. Like in the previous example, the original, non-IFT, RSA core is part of the benchmarks used by Salmani et al. [29] and Shakya et al. [30] and can be found on Trust-Hub [28]. Using Chapter 4's procedure, we determined that the bug location consists of 21 registers where tainted timing flow is generated:

1. *rsacypher.count*
2. *rsacypher.multgo*
3. *rsacypher.root*
4. *rsacypher.done*
5. *rsacypher.tempin*
6. *rsacypher.sqrin*
7. *rsacypher.cypher*
8. *rsacypher.modreg_sqrt*
9. *rsacypher.modreg_mult*
10. *rsacypher.modsqr.first*
11. *rsacypher.modsqr.mpreg*
12. *rsacypher.modsqr.mcreg*
13. *rsacypher.modsqr.modreg*1
14. *rsacypher.modsqr.modreg*2
15. *rsacypher.modsqr.prodreg*
16. *rsacypher.modmultiply.first*
17. *rsacypher.modmultiply.mpreg*
18. *rsacypher.modmultiply.mcreg*
19. *rsacypher.modmultiply.modreg*1
20. *rsacypher.modmultiply.modreg*2
21. *rsacypher.modmultiply.prodreg*

These 21 registers were also the design's unbalanced registers. As in real world conditions, we required in our experiments that the key input be sensitive. Note that 6 other registers existed in the design:

1. *rsacypher.modsqr.prodreg*1
2. *rsacypher.modsqr.prodreg*4
3. *rsacypher.modsqr.prodreg*4
4. *rsacypher.modmultiply.prodreg*1
5. *rsacypher.modmultiply.prodreg*4
6. *rsacypher.modmultiply.mcreg*2

However, we considered these registers to be wires as all of their assignments were confined to `always @(*)` blocks, thus making them part of combinational logic. We therefore did not track their *is_generation* values. In this example, we monitored the timing labels for the data output (*cypher_time*) and ready (*ready_time*) signals. We also utilized five random inputs to run the technique on.

5.2.6 Direct Mapped Cache

This direct mapped cache exhibits timing-based leakage in the output when a data access's address, specifically the part corresponding to the cache index, is labeled as functionally sensitive and the cache encounters a stall. Note that the original, non-IFT, cache was implemented by the authors of Clepsydra [1]. Any subsequent accesses to the same line will continue to result in tainted timing flow in the output data. In this example, we want to find the source of that timing-based leakage. The exact bug location, found through the procedure in Chapter 4, are the registers: *cache.MyCtrl.cache_we*, *cache.MyCtrl.stall_cycles*, *cache.MyCtrl.wr_cache_line_enabled*, and *cache.Mem*32*.mem*. We utilized multiple faulty inputs with the same general structure. The inputs first requested a data write whose address's index was functionally tainted, they then requested a no-operation (write/read request register set to 0 for some cycles), and they finally requested a read from the same cache set as the written data, but with a different address. We monitored only the cache's read data output's label called *rd_dat_proc_time* and ignored all other output labels. In the design, 6 registers were unbalanced.

5.2.7 Experimental Setup

The examples we tested our technique on were coded and instrumented with IFT in Verilog, which we compiled and simulated with Icarus Verilog [35]. We then proceeded to do the following for each example:

1. Executed the Pre-Processing step by inserting logging statements for every monitored label-affecting register's timing label at every cycle. We also inserted *is_generation* labels and the corresponding tracking logic for each unbalanced register.

2. Conducted the Simulation and Identification phase by simulating the design on faulty inputs and then using a Python script to analyze the simulation traces.

5.3 Accuracy Evaluation

In this section, we first present the Taint Generation Logging Technique's candidate bug location and accuracy for each example. We then summarize the results. Finally, we analyze this technique's accuracy on handmade examples and identify major sources of false positives not found in the examples.

5.3.1 Multiplier

In this example, the faulty simulations executed for a total of 9 cycles. As we can see in Table 5.1, only two registers out of the eight that affect the monitored labels ever generated tainted timing flow: $qmults.reg\_overflow$ and $qmults.reg\_working\_result$. These two registers were therefore accurately proposed as candidates after the Simulation and Candidate Identification Phase.

Table 5.1: The IFT timing behavior of monitored label affecting registers in the multiplier example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow.

Register	Carries Tainted Timing Flow?	Is Generation?
qmults.reg_overflow	Yes	Yes
qmults.reg_working_result	Yes	Yes
qmults.reg_working_result_fix	No	No
qmults.reg_multiplier_temp	No	No
qmults.reg_multiplicand_temp	No	No
qmults.reg_count	No	No
qmults.reg_done	No	No
qmults.reg_sign	No	No

5.3.2 Divider

Table 5.2: The IFT timing behavior of monitored label affecting registers in the divider example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow.

Register	Carries Tainted Timing Flow?	Is Generation?
qdiv_orig_ift.reg_quotient_fix	Yes	No
qdiv_orig_ift.reg_working_quotient	Yes	Yes
qdiv_orig_ift.reg_working_dividend	Yes	Yes
qdiv_orig_ift.reg_overflow	Yes	Yes
qdiv_orig_ift.reg_working_divisor	No	No
qdiv_orig_ift.reg_count	No	No
qdiv_orig_ift.reg_done	No	No
qdiv_orig_ift.reg_sign	No	No

During faulty simulations, the divider executed on average for 11.00 cycles. Of the eight registers in the design (which also affect the monitored labels), four registers carried tainted flow: $qdiv\_orig\_ift.reg\_quotient\_fix$, $qdiv\_orig\_ift.reg\_working\_quotient$, $qdiv\_orig\_ift.reg\_working\_dividend$, and $qdiv\_orig\_ift.reg\_overflow$ (Table 5.2). Of those four registers, $qdiv\_orig\_ift.reg\_working\_quotient$, $qdiv\_orig\_ift.reg\_working\_dividend$, $qdiv\_orig\_ift.reg\_overflow$ generated tainted timing flow and were accurately proposed by the Taint Generation Logging Technique as candidates.

Table 5.3: The IFT timing behavior of monitored label affecting registers in the AES with divider - base example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow.

Register	Carries Tainted Timing Flow?	Is Generation?
aes_128.a5.out_1	Yes	No
aes_128.a6.k3a	Yes	No
aes_128.a6.S4_0.S_2.out	Yes	No
aes_128.r6.state_out	Yes	No
aes_128.a6.out_1	Yes	No
aes_128.r7.t3.t3.s4.out	Yes	No
⋮	⋮	⋮
aes_128.a10.S4_0.S_3.out	Yes	No
aes_128.a10.S4_0.S_2.out	Yes	No
aes_128.a10.S4_0.S_1.out	Yes	No
aes_128.a10.S4_0.S_0.out	Yes	No
aes_128.rf.state_out	Yes	No
aes_128.a10.S4_0.S_0.out	Yes	No
aes_128.a5.qdiv_orig_ift.reg_quotient_fix	Yes	No
aes_128.a5.qdiv_orig_ift.reg_working_quotient	Yes	Yes
aes_128.a5.qdiv_orig_ift.reg_working_dividend	Yes	Yes
aes_128.s0	No	No
aes_128.k0	No	No
⋮	⋮	⋮
aes_128.a1.S4_0.S_1.out	No	No
aes_128.a1.S4_0.S_0.out	No	No

5.3.3 AES Cores Injected with Divider - Base Example

The faulty simulations in this example ran for 32.00 cycles. Of the 413 registers in this design that affect the monitored labels, 134 carried tainted flow. The middle-lower portion of Table 5.3 contains simulation information about bug location relevant registers. Only *aes_128.a5.qdiv_orig_ift.reg_working_quotient* and *aes_128.a5.qdiv_orig_ift.reg_working_dividend* generated tainted timing flow and thus were accurately proposed as candidates by the Taint Generation Logging Technique.

Table 5.4: The IFT timing behavior of monitored label affecting registers in the AES with divider - Loop example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow.

Register	Carries Tainted Timing Flow?	Is Generation?
aes_128.a5.out_1	Yes	No
aes_128.a6.k3a	Yes	No
aes_128.a6.S4_0.S_2.out	Yes	No
aes_128.a2.k3a	Yes	No
aes_128.a2.S4_0.S_2.out	Yes	No
aes_128.r6.state_out	Yes	No
aes_128.r2.state_out	Yes	No
aes_128.a6.out_1	Yes	No
aes_128.a2.out_1	Yes	No
aes_128.r7.t3.t3.s4.out	Yes	No
⋮	⋮	⋮
aes_128.rf.state_out	Yes	No
aes_128.a10.S4_0.S_0.out	Yes	No
aes_128.a5.qdiv_orig_ift.reg_quotient_fix	Yes	No
aes_128.a5.qdiv_orig_ift.reg_working_quotient	Yes	Yes
aes_128.a5.qdiv_orig_ift.reg_working_dividend	Yes	Yes
aes_128.s0	No	No
aes_128.k0	No	No
⋮	⋮	⋮
aes_128.a1.S4_0.S_1.out	No	No
aes_128.a1.S4_0.S_0.out	No	No

5.3.4 AES Cores Injected with Divider - Loop Example

The faulty simulations in this example ran for 45 cycles. During the faulty simulations, the tainted timing flow generated in the divider module looped from $a5$ to $a2$, which in turn tainted a large proportion of earlier registers' labels. As a result, 270 of the 413 registers that affect the monitored labels carried tainted flow. Part of Table 5.4 displays the bug location relevant registers and some other registers that carried tainted flow. It shows that earlier registers such as $aes\_128.a2.out\_1$ carried tainted flow. Similar to the base example, $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_quotient$ and $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_dividend$ generated tainted flow. They were thus the candidate set, which accu-

Table 5.5: The IFT timing behavior of monitored label affecting registers in the AES with divider - Timing Blockage example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow.

Register	Carries Tainted Timing Flow?	Is Generation?
aes_128.a5.qdiv_orig_ift.reg_working_quotient	Yes	Yes
aes_128.a5.qdiv_orig_ift.reg_working_dividend	Yes	Yes
aes_128.a5.qdiv_orig_ift.reg_quotient_fix	Yes	No
aes_128.a5.out_1	Yes	No
aes_128.a6.k3a	Yes	No
⋮	⋮	⋮
aes_128.a10.k3a	Yes	No
aes_128.a10.S4_0.S_3.out	Yes	No
aes_128.a10.S4_0.S_2.out	Yes	No
aes_128.a10.S4_0.S_1.out	Yes	No
aes_128.a10.S4_0.S_0.out	Yes	No
aes_128.rf.state_out	Yes	No
aes_128.a2.qdiv_fix_ift.reg_working_quotient	Yes	Yes
aes_128.a2.qdiv_fix_ift.reg_working_dividend	Yes	Yes
aes_128.a2.qdiv_fix_ift.reg_quotient_fix	No	No
aes_128.k0	No	No
⋮	⋮	⋮
aes_128.a1.S4_0.S_1.out	No	No
aes_128.a1.S4_0.S_0.out	No	No

rately predicted the bug location.

5.3.5 AES Cores Injected with Divider - Timing Blockage Example

The simulations in this example ran for 43 cycles. Table 5.5 contains the generation and taint carrying behavior of the registers in this design. The set of taint generating were the same as those in the AES with divider base example, but with two additions: *aes_128.a2.qdiv_fix_ift.reg_working_quotient* and *aes_128.a2.qdiv_fix_ift.reg_working_dividend*. Therefore the Taint Generation Logging Technique incorrectly included *aes_128.a2.qdiv_fix_ift.reg_working_quotient* and *aes_128.a2.qdiv_fix_ift.reg_working_dividend* in its candidate bug location. The tainted flows generated at these two additional registers were always blocked by *aes_*

Table 5.6: The IFT timing behavior of monitored label affecting registers in the RSA example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow.

Register	Carries Tainted Timing Flow?	Is Generation?
rsacypher.root	Yes	Yes
rsacypher.modreg_mult	Yes	Yes
rsacypher.tempin	Yes	Yes
rsacypher.sqrin	Yes	Yes
rsacypher.modreg_sqrt	Yes	Yes
rsacypher.multgo	Yes	Yes
rsacypher.modsqr.first	Yes	Yes
rsacypher.modsqr.mpreg	Yes	Yes
rsacypher.modsqr.mcreg	Yes	Yes
rsacypher.modsqr.modreg1	Yes	Yes
rsacypher.modsqr.modreg2	Yes	Yes
rsacypher.modsqr.prodreg	Yes	Yes
rsacypher.modmultiply.first	Yes	Yes
rsacypher.modmultiply.mpreg	Yes	Yes
rsacypher.modmultiply.mcreg	Yes	Yes
rsacypher.modmultiply.modreg1	Yes	Yes
rsacypher.modmultiply.modreg2	Yes	Yes
rsacypher.modmultiply.prodreg	Yes	Yes
rsacypher.count	Yes	Yes
rsacypher.cypher	Yes	Yes
rsacypher.done	Yes	Yes

128.*a2.qdiv_fix_ift.reg_quotient_fix*, meaning that they were disjointed registers. This result indicates that this technique suffers from disjointed taint generating register false positives when there are timing flow blockage points.

5.3.6 RSA

The RSA simulations ran for an extremely long time compared to the runtime of other tested examples: 16576.6 cycles on average. Moreover, all of the unbalanced registers carried and generated tainted flow. The candidate set identified by the Taint Generation Logging Technique was therefore all 21 unbalanced registers in the design. The Taint Generation Logging Technique therefore accurately found the bug location in this example.

Table 5.7: The IFT timing behavior of monitored label affecting registers in the cache example (data collected by our techniques). Carries Tainted Timing Flow? = Whether the register's timing label was tainted, Is Generation? = Whether the register generated tainted timing flow.

Register	Carries Tainted Timing Flow?	Is Generation?
cache.MyCtrl.cache_we	Yes	Yes
cache.MyCtrl.stall_cycles	Yes	Yes
cache.MyCtrl.wr_cache_line_enabled	Yes	Yes
cache.Mem32.mem	Yes	Yes
cache.MyCtrl.rst_cache	No	No

5.3.7 Cache

The faulty simulations on the cache example ran for 240.00 cycles. After analysis on the traces, out of the five registers that affect the monitored label, the Taint Generation Logging Technique found four registers that had generated tainted timing flow: *cache.MyCtrl.cache_we*, *cache.MyCtrl.stall_cycles*, *cache.MyCtrl.wr_cache_line_enabled*, and *cache.Mem32.mem* (Table 5.7). This technique therefore accurately identified those four registers as the bug location.

5.3.8 Experiments' Summary

In all examples without blockage points, the Taint Generation Logging Technique localized their bugs with 100% accuracy. However, when timing flow blockage points were introduced, the technique suffered from false positives in the form of disjointed registers. We now analyze the technique with handmade examples to explore more false positive sources.

5.3.9 Accuracy Analysis on Handmade Examples

While the Taint Generation Logging Technique was able to localize bugs with 100% accuracy on example designs without blockage points, those designs were too small and simple to expose the false positives that are not caused by timing flow blockage points. In differently structured designs, the technique will produce disjointed register false positives whose sources

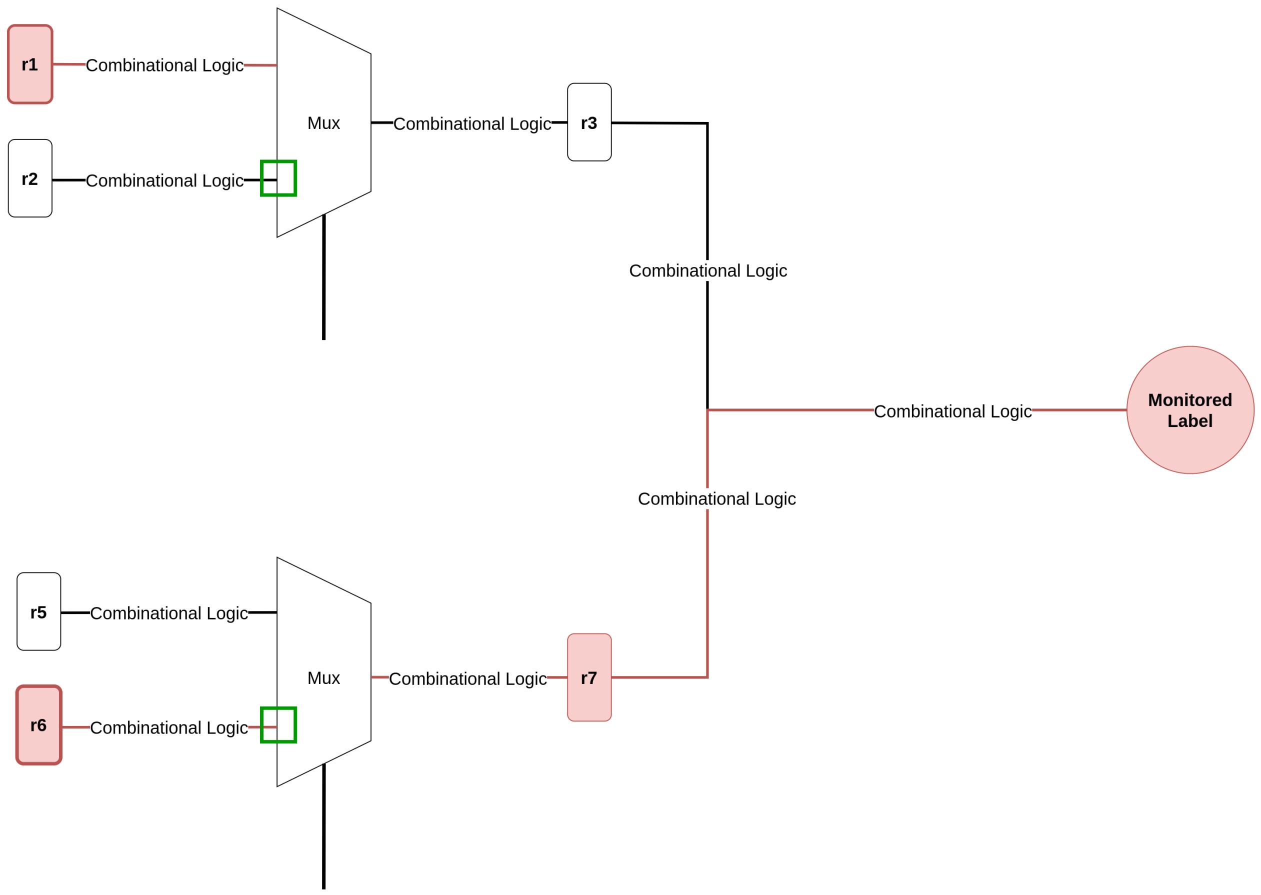


Figure 5.1: A faulty simulation on a theoretical design that results in disjointed register false positives in the Taint Generation Logging Technique's candidate set. Each rounded rectangle represents a register and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box.

are not blockage points. For example, consider the hardware design simulation found in Figure 5.1. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box. If a design only has faulty simulations like the one in Figure 5.1, then $r1$ would be a disjointed register that generates tainted flow. The Taint Generation Logging Technique would incorrectly suggest $r1$ as part of the bug location since it generates tainted flow. Consider another example in Figure 5.2. Again, if the design only has faulty simulations like the two in Figure 5.2, then the control flow behavior at both multiplexers

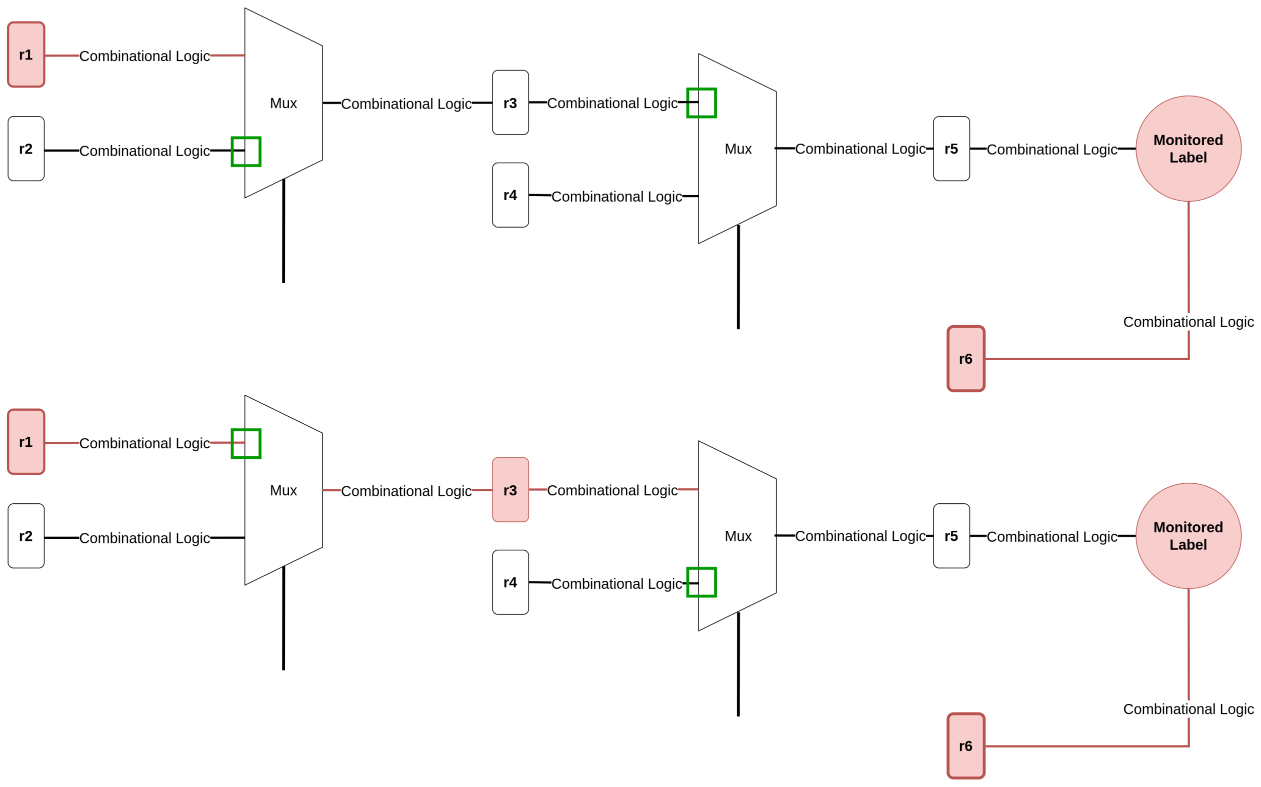


Figure 5.2: Two faulty simulations (a) and (b) on another theoretical design that result in disjointed register false positives in the Taint Generation Logging Technique's candidate set. Each rounded rectangle represents a register and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box.

would result in $r1$ being proposed as a bug location even though its a disjointed register. If faulty simulations on designs contain multiplexer behavior similar to that of the examples in Figure 5.1 and 5.2, then disjointed taint generating registers will be produced and the Taint Generation Logging Technique will incorrectly identify them as the bug location. This control flow/multiplexer behavior and design structuring is very common in hardware designs. For example, many designs have execution modes, which could cause a multiplexer to always select one input during faulty simulations. This technique therefore cannot always achieve high accuracy localization on more complex or differently structured designs even when they don't possess timing flow blockage points.

5.4 Efficiency Evaluation

We now informally reason about the Taint Generation Logging Technique's worst-case time and space complexities. Again note: we do not claim that this analysis is precise. Rather, it provides a general idea of the technique's efficiency. We define the following variables for our complexity analysis:

- s = number of simulations.
- u = number of unbalanced registers that affect the monitored labels. Note that the number of taint generating registers is less than or equal to u.
- r = number of registers that affect the monitored labels.
- c = number of cycles during the simulation.
- n = the maximum number of operations during an unmodified simulation.

We define the following costs for our complexity analysis:

- $O(1)$ = the worst-case time cost of executing one of a simulation's operations at a cycle (e.g. IFT logic, register assignments, and combinational logic assignments). Note then, that the worst-case time complexity of a simulation is $O(n)$ and the time complexity of running all simulations is $O(sn)$.
- $O(1)$ = the time complexity of retrieving and comparing a trace matrix's entry.
- $O(x)$ = the maximal memory footprint of an unmodified simulation.
- $O(1)$ = the memory footprint of a trace matrix's entry.

Furthermore, for simplicity, we assume that the action of logging the simulations' behavior (registers, IFT, control signals) takes $O(1)$ time and space.

5.4.1 Time Complexity

During simulation, this technique tracks for tainted flow generation, which at worst, adds another taint tracking logic for each register. The number of operations is therefore at most, a constant times the number of operations in an unmodified simulation. Therefore, the Taint Generation Logging Technique's simulation runtime is at worst $O(sn)$. The localization part of the technique needs to search the high-low values for each register at each cycle for each simulation, which results in a worst-case complexity of $O(src) + O(suc) = O(src)$ (since $u \leq r$). Thus, the overall worst-case complexity is $O(sn) + O(src) = O(s(n + rc))$.

5.4.2 Space Complexity

The memory overhead during each simulation is an additional taint generation tracking label for each unbalanced register. This additional memory is at most a constant times the original memory footprint. Thus, the complexity is $O(x)$. Again note that the complexity is not $O(sx)$ because the simulation's memory can be freed after completion. During the localization portion of the technique, the trace matrices have to be loaded, which results in a worst-case space complexity of $O(src) + O(suc) = O(src)$ (since $u \leq r$). The overall worst-case complexity is therefore $O(x + src)$.

CHAPTER 6
BRUTE-FORCE BACKTRACKING

The Brute-Force Backtracking Technique automates backtracking along the design's control flow/data dependency graph for each simulation. We now provide a short description of how this technique functions.

6.1 Description

This technique utilizes faulty simulation traces to walk backwards along all insecure paths to the bug location. Note that this technique also considers time during its backwards walk. At a high level, for each simulation, the technique starts at the monitored labels and then executes the following procedure:

1. Let S be the set of signals that the technique is currently at. Walk backwards along the register dependency/control flow graph to adjacent registers that, in the previous cycle, carried tainted flow that was not blocked from propagating to signals in S. Call this set of registers R_a.

2. If R_a is empty, then terminate. Else, continue to step 3.

3. A register along an insecure path may also generate tainted flow (even if it is not at the beginning). For each $r \in R_a$, check if it generated tainted flow at the current cycle. If so, then add r to the bug location set.

4. Return to step 1 if the current cycle is not the simulation's first cycle. If the current cycle is cycle 1, then terminate.

The implementation of this technique views each simulation as a collection of m c deep data flow trees, where m is the number of monitored labels and c is the number of cycles in that simulation. Note that the nodes in this tree are uniquely identified by register-cycle pairs.

That is, each node represents a register's state at a certain cycle. We allow registers to appear more than once in the tree in order to "roll out" data loops. The technique then localizes the timing bug by executing a modified Depth-First or Breadth-First Search (DFS or BFS) algorithm over each of the m trees. More specifically, for each tree, the technique starts at a monitored label and terminates when it finds all insecure paths that connect to that label.

6.2 Accuracy Evaluation

This technique clearly achieves 100% accuracy on all examples. Since the technique traverses all tainted timing flow propagation paths that end at the monitored labels, it will eventually find all bug location registers. We therefore do not conduct an in-depth accuracy analysis for this technique.

6.3 Efficiency Evaluation

We now reason about this technique's worst-case time and space complexities. We define the following variables for our complexity analysis:

- s = number of simulations.
- u = number of unbalanced registers that affect the monitored labels. Note that the number of taint generating registers is less than or equal to u.
- r = number of registers that affect the monitored labels.
- c = number of cycles during the simulation.
- b = the maximum branching factor of the "rolled out" data flow tree traversed by the DFS or BFS algorithm.
- n = the maximum number of operations during an unmodified simulation.

We define the following costs for our complexity analysis:

- $O(1)$ = the worst-case time cost of executing one of a simulation's operations at a cycle (e.g. IFT logic, register assignments, and combinational logic assignments). Note then, that the worst-case time complexity of a simulation is $O(n)$ and the time complexity of running all simulations is $O(sn)$.
- $O(1)$ = the time complexity of traversing from node to edge and edge to node.
- $O(x)$ = the maximal memory footprint of an unmodified simulation.
- $O(1)$ = the memory footprint of an edge or node in the trees.

Furthermore, for simplicity, we assume that the action of logging the simulations' behavior (registers, IFT, control signals) takes $O(1)$ time and space.

6.3.1 Time Complexity

This technique does not modify the design and thus, the simulation part's runtime is $O(sn)$. The localization part of the technique is very expensive. In the worst-case, the technique needs to conduct a DFS or BFS on potentially $O(b^c)$ registers for each of the m rolled out trees (once for each monitored label). Moreover, the technique has to conduct this search for every faulty simulation. Thus, the worst-case complexity of the localization part is $O(smb^c)$, meaning that the overall worst-case complexity is $O(sn) + O(smb^c) = O(s(mb^c + n))$.

6.3.2 Space Complexity

The memory overhead during simulation is $O(x)$ because the technique does not modify the design. The reason for why the complexity is not $O(sx)$ is the same as the previous two techniques'. The trees that are searched over during localization require a total of $O(mb^c)$ space (for the nodes and edges). $O(mbc)$ space is needed for the DFS/BFS algorithm (well-known complexity for DFS/BFS on large graphs). The overall worst-case complexity is therefore $O(x + mb^c + mbc) = O(x + mb^c)$.

CHAPTER 7
AGGREGATED GRAPH WALKING

The Aggregated Graph Walking (AGW) Technique localizes timing bugs by first aggregating the IFT behavior of all faulty simulations and then using a graph walking algorithm to generate a register dependency graph containing all insecure paths and buggy registers. This technique consists of four phases: Pre-Processing, Simulation and Candidate Identification, Graphing, and Candidate Precision Improvement. Note that this technique's Pre-Processing and Simulation and Candidate Identification phases are exactly the same as the Taint Generation Logging Technique's. We therefore begin our description with the Graphing Phase. Moreover, we draw this section's terminology from the Taint Generation Logging Technique's definitions.

7.1 Description

The AGW Technique first runs the Taint Generation Logging Technique and gets the candidate set R' and taint carrying register set R^*. These sets represent the aggregation of IFT behavior of all faulty simulations. The AGW Technique then executes the Graphing phase.

7.1.1 Graphing

In order to provide the designer with context and better accuracy for the bug location, the AGW Technique constructs a data dependency graph that models all timing flow relationships among the registers that carry tainted flow.

Since the data dependency graph contains all registers with tainted flow, it contains all registers along an insecure path. Now as observed in Chapter 5, the set R' may contain disjointed false positive registers. This technique attempts to eliminate these false positives by constructing the graph with a walking algorithm that starts at the monitored labels and then walks backwards to adjacent taint carrying registers. In this graph, there always exists a path of taint carrying registers from any node to a monitored label, meaning that there is a lesser chance that a taint generating register in the graph is actually disjointed. Note: we say "lesser" chance because the dependency graph doesn't ignore all taint generating disjointed registers. There exist some edge cases that cannot be handled by the graph walking algorithm. We further elaborate on this walking technique's inaccuracy in the Accuracy Evaluation Section below.

For each signal that is one of the monitored labels, if that signal is a register, add it to the graph. For each monitored label that is not a register, identify its adjacent registers (that also affect the monitored label) and add them to the graph. Let this set of initial registers in the graph be $R_{initial}$. As a reminder, adjacency in this context depends on the timing flow label relationships between a signal and other registers. The graph we are about to construct is conceptually a subset of the entire design's register timing label data dependency graph. Now for each $r \in R_{initial}$,

1. Determine all of r's adjacent registers. Let this set be R_a. For each $r_a \in R_a$, check if r_a's timing label affects r's timing label. If r_a does affect r's timing label, then add r_a to R_d. R_d is therefore the set of adjacent registers to r whose timing labels affect r's timing labels.

2. For each $r_d \in R_d \cap R^*$, add r_d and a directed edge outgoing from r and incoming to r_d to the graph. This edge means that r's timing flow label depends on r_d's. In addition, if r_d has not been visited before, add r_d to a global queue.

3. If the queue is empty, then terminate. Otherwise, repeat steps 1 to 3 again for each register in the queue.

The end result is a graph containing all of the insecure paths and ignoring some, if not most, disjointed registers. Moreover, the starting nodes of the insecure paths correspond to taint generating registers. Note that since registers can propagate and generate flow at the same time, these insecure paths may overlap and thus the dependency graph may contain data loops or circular dependencies.

Output: The register dependency graph.

7.1.2 Candidate Precision Improvement

Once the dependency graph has been generated, it should contain the insecure paths and few, if any, disjointed registers. However, the candidate set of registers, R', may still contain previously eliminated disjointed registers that generate tainted timing flow. Thus, the technique eliminates disjointed registers from the candidate set by selecting only the candidate registers that reside in the dependency graph. The technique then proposes this filtered set as the bug location. It also provides the designer with the register dependency graph.

Output: The filtered candidate set of taint generating registers. The register dependency graph.

7.2 Experimental Methodology

Like in Chapter 5, we evaluated this technique's accuracy both through empirical experiments and qualitative exploration on handmade and theoretical examples. For the empirical experiments, we utilized exactly the same examples. Moreover, we utilized the same experimental setup for the first two phases of this technique. We implemented the final two phases of this technique by:

1. Using PyVerilog (including its provided examples), PyGraphviz, custom Python scripts, and the simulation traces to generate the dependency graph [31, 11].

2. Using manual analysis to find the intersection between the set of registers in the dependency graph and the Simulation and Identification phase's candidate set.

7.3 Accuracy Evaluation

In this section, we first present the AGW Technique's dependency graph and accuracy for each example. Note that the monitored signal in all of our examples was a wire and therefore the register dependency graphs do not contain the monitored signal. In order to enhance the reader's understanding, we outlined the nodes that the monitored signal is adjacent to and depends on in red. Those registers can be seen as the "endpoints" of the insecure paths. Moreover, we highlighted in green the labels of the nodes that corresponded to the AGW technique's suggested bug locations. Finally note that the taint generation and carrying behavior of the examples' registers can be found in corresponding tables in Chapter 5. After we present these results, we then summarize them. Finally, we analyze this technique's accuracy on handmade examples and identify major sources of false positives not found in the examples.

7.3.1 Multiplier

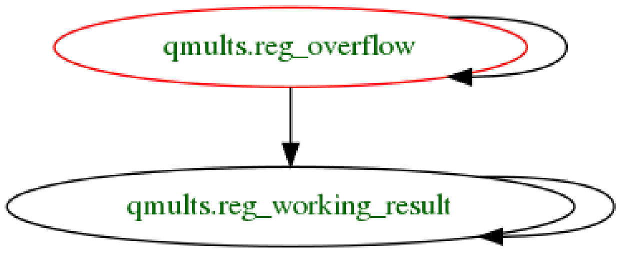


Figure 7.1: The multiplier example's register dependency graph.

The two registers proposed in Chapter 5.3.1 were the only ones to carry tainted flow, which means that the dependency graph in Figure 7.1 only has two registers. Thus, during the Candidate Precision Improvement Phase, the AGW Technique did not filter out any registers from the candidate set and accurately proposed the two registers *qmults.reg_overflow* and *qmults.reg_working_result* as the bug location.

7.3.2 Divider

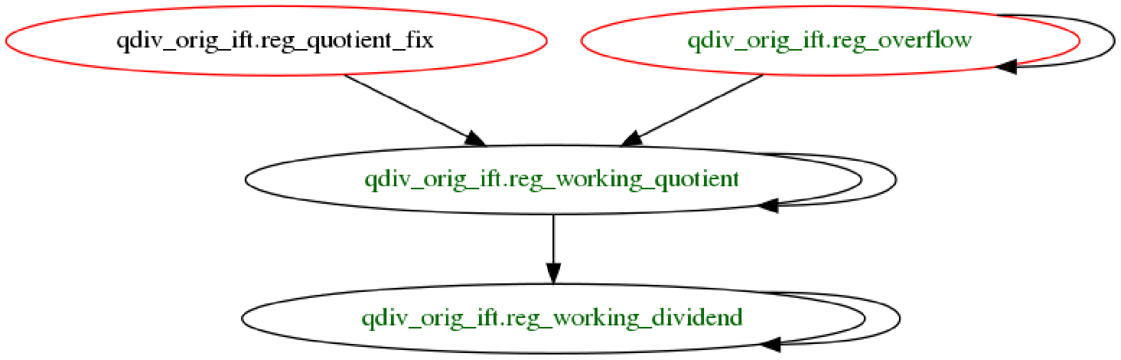

qdiv_orig_ift.reg_quotient_fix
qdiv_orig_ift.reg_overflow
qdiv_orig_ift.reg_working_quotient
qdiv_orig_ift.reg_working_dividend

Figure 7.2: The divider example's register dependency graph.

The AGW technique utilized the four taint carrying registers in Chapter 5.3.2 to generate the dependency graph found in Figure 7.2. As *qdiv_orig_ift.reg_working_quotient*, *qdiv_orig_ift.reg_working_dividend*, and *qdiv_orig_ift.reg_overflow* were in the dependency graph, no registers were filtered out of the candidate set during the last phase. Our technique therefore accurately proposed the bug location as the set of registers *qdiv_orig_ift.reg_overflow*, *qdiv_orig_ift.reg_working_quotient*, and *qdiv_orig_ift.reg_working_dividend*.

7.3.3 AES Cores Injected with Divider - Base Example

Figure 7.3 shows the bug location part of the dependency graph generated from the 134 taint carrying registers found in Chapter 5.3.3 as the entire graph is too large and complex to be displayed. Since the candidates were part of the dependency graph, no registers were filtered out of the candidate set during the Candidate Precision Improvement phase. The technique therefore identified *aes_128.a5.qdiv_orig_ift.reg_working_quotient* and *aes_128.a5.qdiv_orig_ift.reg_working_dividend* as the bug location. Thus, the AGW accurately determined the bug location of this design.

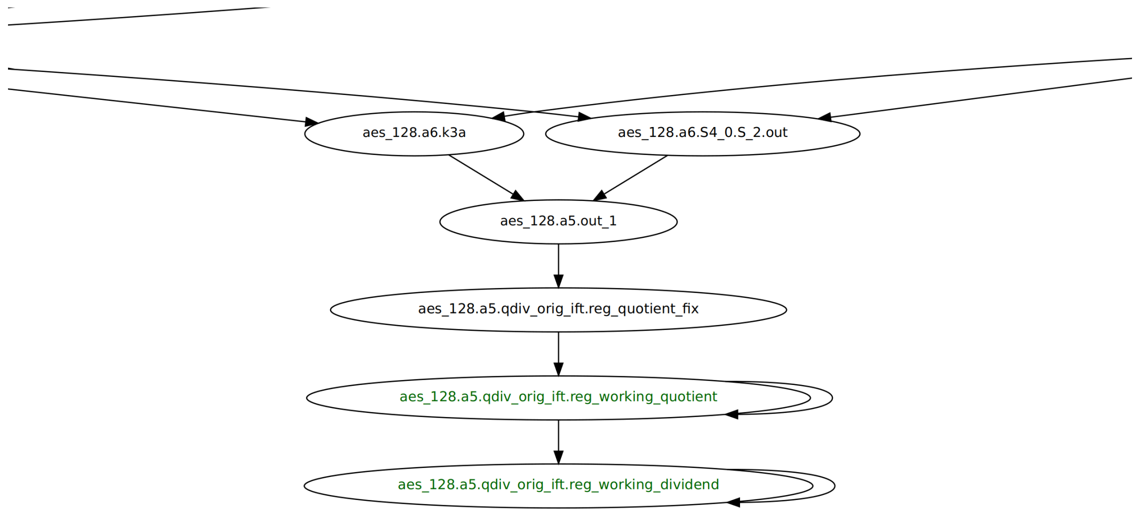

aes_128.a6.k3a
aes_128.a6.S4_0.S_2.out
aes_128.a5.out_1
aes_128.a5.qdiv_orig_ift.reg_quotient_fix
aes_128.a5.qdiv_orig_ift.reg_working_quotient
aes_128.a5.qdiv_orig_ift.reg_working_dividend

Figure 7.3: The AES with divider base example's register dependency graph (truncated).

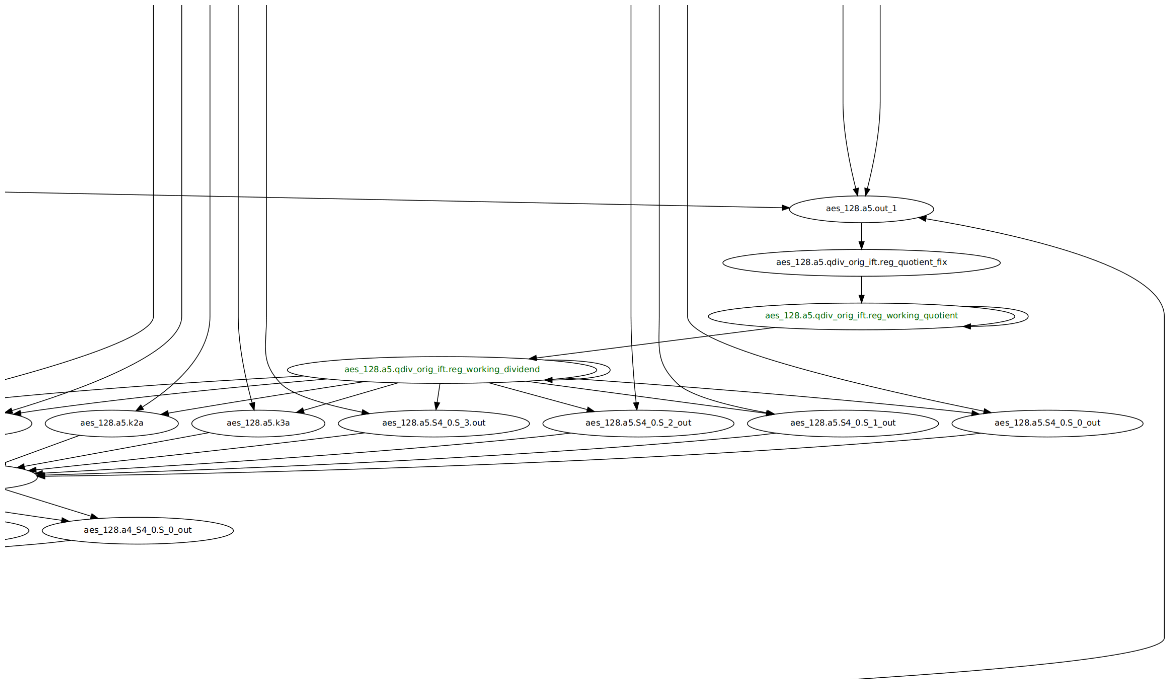


Figure 7.4: The AES with divider - Loop example's register dependency graph (truncated).

7.3.4 AES Cores Injected with Divider - Loop Example

The AGW technique generated the dependency graph found in Figure 7.4, which displays the bug location part of the dependency graph as the original graph is too large and complex for display. Note that $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_dividend$ now depends on a register because of the data loop. During Candidate Precision Improvement, the two registers in the candidate set were checked against the graph and both were found to reside in the graph. Thus, the two registers, $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_quotient$ and $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_dividend$, were identified as the bug location. Thus, our technique accurately determined the bug location of this design.

7.3.5 AES Cores Injected with Divider - Timing Blockage Example

The AGW Technique generated the same dependency graph as the base example's dependency graph (Figure 7.3) because there was at least one untainted register between the disjointed registers and an insecure path. The candidate set the technique thus generated was exactly the same as the set found in the AES with divider base example. This graph walking technique, unlike the Taint Generation Logging Technique, therefore achieved 100% accuracy in an example with blockage points.

7.3.6 RSA

The AGW Technique generated a dependency graph that contained all 21 bug location registers. Figure 7.5 presents a very complex dependency graph laden cycles. These cycles imply that there are many insecure paths, which makes sense because there are twenty-one taint generating registers. The technique checked the set of candidate registers against the graph and filtered out none in the Candidate Precision Improvement phase. Thus, our technique accurately selected all 21 registers as the bug location. Note that we did not include the 6 other registers in the graph since, as discussed earlier in Section 5.2.5, we considered them to be wires/combinational logic.

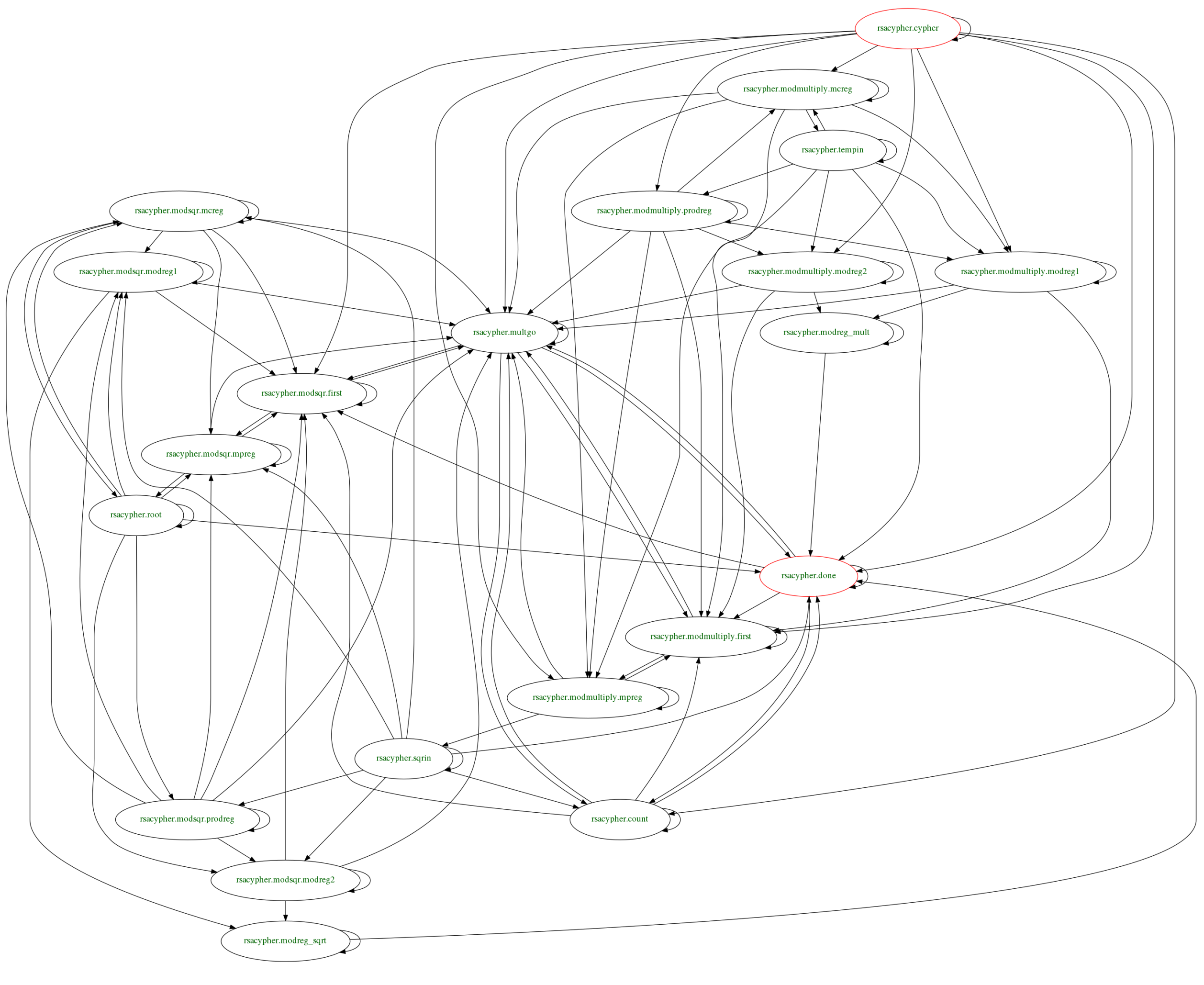


Figure 7.5: RSA example's register dependency graph.

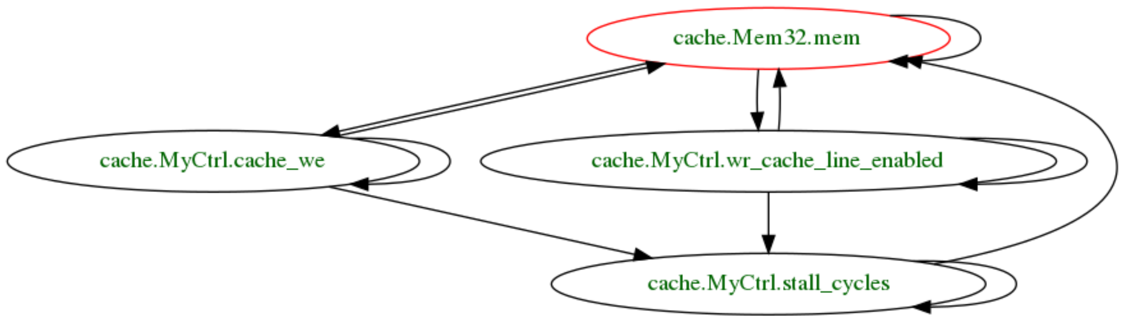


Figure 7.6: Cache example's register dependency graph.

7.3.7 Cache

The dependency graph generated by the AGW Technique shown in Figure 7.6 contains four taint generating registers with a multitude of cycles, indicating overlapping insecure paths.

These taint generating registers are exactly the bug location registers identified in Chapter 5.3.7. No registers were filtered out from the candidate set during the Candidate Precision Improvement Phase. This technique therefore accurately selected the four registers in the candidate set as the bug location.

7.3.8 Experiments' Summary

In all examples without blockage points, the AGW localized their locations with 100% accuracy. However, we also found that the graph generation portion of the technique was unneeded for those examples. There were no disjointed false positives caused by multiplexer behavior. When timing flow blockage points were introduced, the technique was able to eliminate the disjointed register false positives from the candidate set. We now analyze the technique with handmade examples to expose more false positive sources.

7.3.9 Accuracy Analysis on Handmade Examples

First observe that the AGW Technique filters out the disjointed register false positives caused by the multiplexer control flow behavior found in Figures 5.1 and 5.2. That is, if a multiplexer produces a disjointed register d such that there exists a untainted register between d and every register along an insecure path, the graph walking algorithm ignores d. This is because the algorithm begins on an insecure path and walks to adjacent taint carrying registers that its current register depends on. Thus in many cases, the graph walking technique will have better accuracy than the Taint Generation Logging Technique. We did not see such an improvement in the non-blockage experimental examples because their simulations produced no disjointed registers. The examples we utilized were too simple to induce false positives caused by multiplexer behaviors.

While the AGW Technique can be more accurate than the Taint Generation Logging Technique in examples like those in Figures 5.1 and 5.2, it is not completely invulnerable to disjointed register false positives caused by control flow behavior at multiplexers. If a

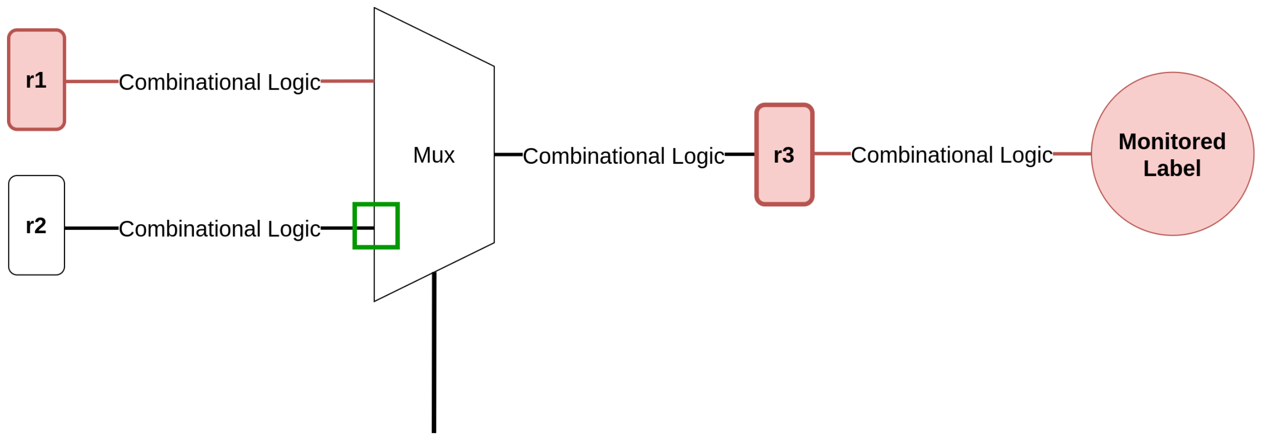


Figure 7.7: A faulty simulation on a theoretical design that results in disjointed register false positives in the AGW Technique's candidate set. Each rounded rectangle represents a register and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box.

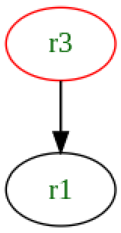


Figure 7.8: Dependency graph produced from the example in Figure 8.3

disjointed register d generates tainted flow that propagates along a path P to a register that is on or adjacent to an insecure path, then the AGW Technique will add any register along P including the disjointed register to the dependency graph. It will also incorrectly propose d as part of the bug location. This problem is a consequence of the aggregated graph, which reduces overall precision, and the graph walking technique's simple walking algorithm. Whenever the technique sees an adjacent register that carries tainted flow, it will walk to that register even if it is a disjointed register.

For example, consider Figure 7.7. If all faulty simulations are like the one in the figure, then the graph walking algorithm will start at $r3$ and walk backwards to $r1$. This is because $r1$ carries tainted flow and is adjacent to and affects $r3$. That is, $r1$ is adjacent to an insecure path. The technique will incorrectly produce the graph found in Figure 7.8 and

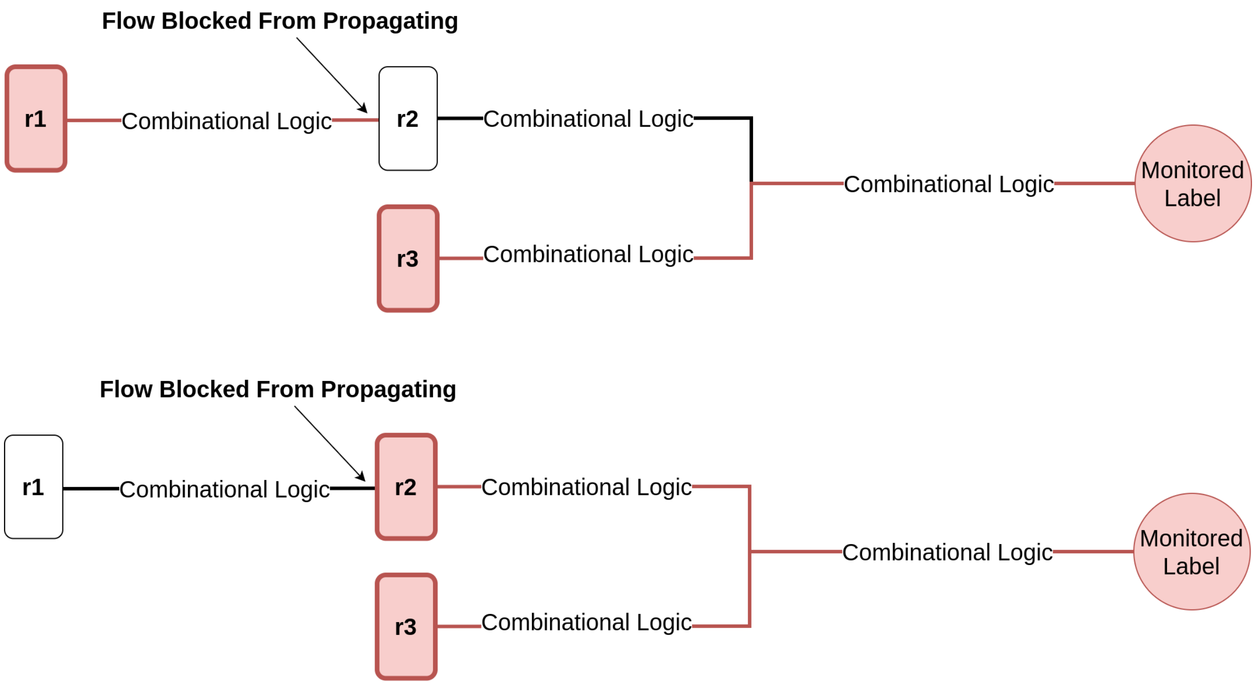


Figure 7.9: Two faulty simulations (a) and (b) on a theoretical design with blockage points that result in disjointed register false positives in the AGW Technique's candidate set. Each rounded rectangle represents a register and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box.

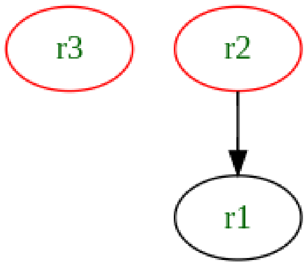


Figure 7.10: Dependency graph produced from the example in Figure 8.5

thus incorrectly propose $r1$ as part of the bug location. Thus, better experimental examples are required to expose these disjointed register false positives caused by multiplexer control flow behavior.

Observe that the AGW technique filtered out disjointed register false positives in the AES with divider and timing flow blockage example. However, the AGW Technique is not impervious to all disjointed registers produced by blockage points. It is vulnerable to disjointed registers whose tainted flow in some simulations gets propagated to a register adjacent to or along an insecure path, but is blocked from propagating onwards to the monitored labels. The graph walking algorithm, which naively walks to adjacent taint carrying registers, is unable to catch these edge cases and will add these disjointed registers to the bug generating location. For example, consider two faulty simulations on a design with a blockage point presented in Figure 7.9. In the first simulation (Figure 7.9 (a)), the tainted flow generated by $r1$ is blocked from propagating to $r2$. The tainted flow generated at $r3$ propagates to the monitored label. In the second simulation (Figure 7.9 (b)), $r1$ no longer generates tainted flow. Tainted flows are generated at $r2$ and $r3$ and then propagate to the monitored labels. The AGW algorithm would produce the dependency graph found in Figure 7.10, which mistakenly identifies $r1$ as part of the bug location. Our timing flow blockage example was too simple to highlight this weakness, suggesting that more insightful and complex examples are required for our empirical evaluations.

7.4 Efficiency Evaluation

For this efficiency evaluation, we define the following variables:

- s = number of simulations.
- u = number of unbalanced registers that affect the monitored labels. Note that the number of taint generating registers is less than or equal to u.
- t = number of taint carrying registers that affect the monitored labels.
- r = number registers that affect the monitored labels.
- c = number of cycles during the simulation.
- n = the maximum number of operations during an unmodified simulation.

We define the following costs for our complexity analysis:

- $O(1)$ = the worst-case time cost of executing one of a simulation's operations at a cycle (e.g. IFT logic, register assignments, and combinational logic assignments). Note then, that the worst-case time complexity of a simulation is $O(n)$ and the time complexity of running all simulations is $O(sn)$.

- $O(1)$ = the time complexity of traversing from node to edge and edge to node in the graph walking algorithm.

- $O(n)$ = the worst-case cost of executing all of a simulation's operations (e.g. register and combinational logic assignments). This number also includes the cost of executing the IFT logic.

- $O(x)$ = the maximal memory footprint of an unmodified simulation.

- $O(1)$ = the memory footprint of an edge or node in the dependency graph.

Furthermore, for simplicity, we assume that the action of logging the simulations' behavior (registers, IFT, control signals) takes $O(1)$ time and space.

7.4.1 Time Complexity

The AGW's simulation runtime is exactly the same as the Taint Generation Logging Technique's runtime because their first two phases are the same. The simulation part's runtime is thus $O(sn)$. The localization part of this technique first executes the Taint Generation Logging Technique's localization portion, which is $O(src)$ time. During the Graphing Phase, the technique needs to conduct graph walk on potentially $O(t)$ registers (nodes) and $O(t^2)$ edges. The complexity is therefore $O(t^2)$. This complexity is not like the Brute-Force Backtracking's exponential complexity because 1) there is a known amount of registers, 2) if our walking algorithm walks to a node that has already been visited, it does not continue to walk from that node, 3) it does not need to traverse the graph for each simulation because it aggregates them in the first two phases. Finally, the Candidate Precision Improvement Phase is at worst $O(ut)$, which is less than or equal to $O(t^2)$. Thus, the worst-case complexity of the localization part is $O(src) + O(t^2)$. The total complexity is therefore $O(s(n + rc) + t^2)$.

7.4.2 Space Complexity

The memory overhead during simulation is $O(x)$ for the same reasons as the Taint Generation Logging Technique's. The graph that is searched over during localization requires $O(t^2)$ space (for the nodes and edges). $O(t)$ space is needed to store registers in a queue during the graph walk. $O(src)$ space is also needed for the localization portion of the Taint Generation Logging Technique. Thus, the localization space complexity is $O(t^2) + O(t) + O(src) = O(t^2 + src)$. The overall worst-case complexity is thus $O(x + t^2 + src)$.

CHAPTER 8
UNIQUE TAINT TRACKING

The Unique Taint Tracking Technique uniquely identifies tainted flows by their originating registers and then implements logic to track them as they propagate to other registers in a design. Each register or monitored label, s, possesses an array that tracks the unique tainted flows currently residing in s's timing label. During simulation with this new logic, only the bug location registers' unique tainted flows propagate to the monitored labels and thus can be quickly and easily identified.

8.1 Description

In this section, we provide a detailed description of this technique. This technique first executes the Taint Generation Logging Technique to determine the set of registers, R', that generate tainted flow during the faulty simulations. Then for each register r in R, a taint tracking array called r_taint_array is added. Next, for each register $r' \in R'$, we generate a unique r'_taint_id. For each of r''s procedural assignments $r' <= u$, this technique adds the following logic to the design:

$$r'\_taint\_array <= ((\text{all controllers are sensitive or not fully controlling})\ ? \tag{8.1}$$

$$u'_0\_taint\_array + u'_1\_taint\_array\ +\ ...\ +\ u'_n\_taint\_array : []) + \tag{8.2}$$

$$(s_{r_0}\_taint\_array\ +\ s_{r_1}\_taint\_array\ +\ ...\ +\ s_{r_n}\_taint\_array) + \tag{8.3}$$

$$((!is\_bal(r')\ \&\ (\text{one of } (r' <= u)\text{'s control signals carries tainted functional flow}))\ ? \tag{8.4}$$

$$[r'\_taint\_id] : []) \tag{8.5}$$

Where:

- The operator "+" represents array concatenation with deduplication.

- $U = \{u'_0, u'_1, ..., u_n\}$ is the set of registers that are adjacent to and affect ("drive") u if u is not a register. If u is a register, then $u'_0 = u$ and $n = 0$.
- $\{u'_0\_taint\_array, u'_0\_taint\_array..., u'_n\_taint\_array\}$ is the set of corresponding unique taint arrays for each register in U.
- $S_{r'} = \{s_{r'_0}, s_{r'_1}, ..., s_{r'_n}\}$ the set of registers that drive the control signals of $r' <= u$.
- Correspondingly, $\{s_{r'_0}\_taint\_array, s_{r'_1}\_taint\_array, ..., s_{r_n}\_taint\_array\}$ is the set of taint arrays for each register in $S_{r'}$.

The added logic is similar to the timing flow tracking logic from registers. The concatenations found in lines 8.1 to 8.3 merge the driving registers' taint arrays into one taint array that contains all of the unique taints that affect r''s timing label. Note that line 8.1 chooses the taint arrays of the registers affecting u only when the tainted flow propagating from u is not blocked (thus handling the blockage case). When a new taint is generated at r' at line 8.4, line 8.5 adds to the taint array a new ID corresponding to that taint. Similarly, for every other register x in the design, the approach adds the following logic for each assignment $x <= u$:

$$x\_taint\_array <= ((\text{all controllers are sensitive or not fully controlling})\ ? \tag{8.6}$$

$$u'_0\_taint\_array + u'_1\_taint\_array\ +\ ...\ +\ u'_n\_taint\_array : []) + \tag{8.7}$$

$$(s_{r_0}\_taint\_array\ +\ s_{r_1}\_taint\_array\ +\ ...\ +\ s_{r_n}\_taint\_array) \tag{8.8}$$

This logic does not include a fourth or fifth line because we know that x does not generate tainted timing flow. Finally, for each monitored label l, a taint array called l_taint_array is added to the design and the following logic is added (if the label is not a register's label):

$$l\_taint\_array <= d_0\_taint\_array\ +\ d_1\_taint\_array\ +\ d_n\_taint\_array \tag{8.9}$$

where $\{d_0\_taint\_array,\ d_1\_taint\_array, ..., d_n\_taint\_array\}$ is the set of taint arrays of the registers that drive the value of the monitored label.

After the technique modifies the design, it then runs the simulations and collects the monitored labels' taint arrays over all simulations. The technique then proposes the set of unbalanced registers residing in this collection of arrays as the bug location.

8.2 Accuracy Evaluation

Like the Selective Taint Generation and Backtracking Techniques, this technique achieves 100% accuracy. Its unique taint tracking logic exactly mirrors the original IFT timing flow logic. By tracking how specific flows propagate to the monitored labels, the technique avoids the explicit tracing of multiplexer and blockage point behavior. If a register's taint does not flow to the monitored label, its unique identifier also never flows to the monitored label. If a register's taint does flow to the monitored label, its unique identifier will flow to the monitored label as well. As this explanation is straightforward, we do not need to further conduct a more detailed accuracy evaluation for this technique.

8.3 Efficiency Evaluation

While this strategy sounds great on paper, it is both time and space inefficient. In the following subsections, we provide an informal worst-case complexity analysis on both dimensions. We define the following variables for our complexity analysis:

- s = number of simulations.
- u = number of unbalanced registers that affect the monitored labels. Note that the number of taint generating registers is less than or equal to u.
- r = number of registers that affect the monitored labels.
- c = number of cycles during the simulation.

- n = the maximum number of operations during an unmodified simulation.

We define the following costs for our complexity analysis:

- $O(1)$ = the worst-case time cost of executing one of a simulation's operations at a cycle (e.g. IFT logic, register assignments, and combinational logic assignments). Note then, that the worst-case time complexity of a simulation is $O(n)$ and the time complexity of running all simulations is $O(sn)$.
- $O(1)$ = the time it takes to retrieve and compare an element in a taint array.
- $O(d)$ = the worst-case time cost of concatenating and deduplicating the arrays over all unique taint tracking operations. We assume that $d >> 1$ and $d >> n$.
- $O(x)$ = the maximal memory footprint of an unmodified simulation.
- $O(1)$ = Each taint array's element's memory footprint.

Furthermore, for simplicity, we assume that the action of logging the simulations' behavior (registers, IFT, control signals) takes $O(1)$ time and space.

8.3.1 Time Complexity

Consider a register r instrumented with the additional taint tracking code. During simulation, concatenating the arrays of two or more signals is extremely slow because new memory may have to be allocated for each concatenation and extra computations are needed to deduplicate elements in the array concatenations. This is the reason why we assume that the concatenation and deduplication of the taint arrays is $O(d)$, where $d >> 1$ and $d >> n$. The simulation complexity is therefore $O(sn + sd) = O(sd)$. During localization, the technique first runs the Taint Generation Logging Technique's localization portion. Unique Taint Tracking then needs to check whether an unbalanced register's taint makes it to the monitored labels. This check requires a search over an array of max size u for every monitored label in every simulation. Therefore, the worst-case time complexity for localization is $O(src+u^2sm)$. The total time complexity is therefore $O(sd)+O(u^2sm+src) = O(sd+u^2sm) = O(s(d+u^2m))$.

8.3.2 Space Complexity

Observe that during simulation, the timing flow label of a register is potentially affected by a multitude of other labels (e.g. control signals' labels). Therefore, the arrays must be prohibitively large in the worst-case. Each register's array may contain u registers at most, and because the worst-case memory footprint of registers is $O(x)$, the worst-case space complexity is $O(ux)$. During localization, this technique needs to explore the monitored labels' arrays over all simulations, which requires at most $O(sum)$ space. It also needs to run the localization portion of the Taint Generation Logging Technique. Thus, the worst-case space complexity is $O(u(sm + x) + src)$.

CHAPTER 9

THE BLOOM FILTER APPROACH

This technique, like the Unique Taint Tracking Technique, tracks the propagation of taints uniquely identified by their originating location throughout a design. However, by allowing for a small false positive rate, this chapter's technique is much more efficient than the previous chapter's. This technique utilizes Bloom filters [1] to track the unique taints residing in registers' timing labels and the monitored labels. A Bloom filter [4] is an array of bits that stores elements for later queries. Bloom, the author of [4], defines the following procedure for element insertions and searches. When an element is added to the filter, d different bit locations in the filter are computed using d hash functions/operations and then set to 1. When that same element is searched for in the filter, the searching entity takes the same d hashed locations and determines that the element is in the filter if the locations all have the value 1. Note that the d hash operations must remain the same for a filter.

Bloom [4] also identifies and analyzes many properties about Bloom filters, several of which we now report in this paragraph. Bloom filters' constant array size makes them vulnerable to false positives during queries. However, they can usually store a multitude of elements before the false positive rate gets too high. Bloom also determines the existence of an optimal d that minimizes the false positive rate given the number of elements in and the size of the filter. Moreover, he derives equations that express the relationships among parameters such as filter size, number of elements, false positive rate, value of d, etc.

We now further identify several more innate properties of Bloom filters based on their description in Bloom's paper [4]. Bloom filters are, by construction, not affected by duplicate additions since the hashing of that element will always return the same bit positions. Moreover, it is clear that we can construct a new filter that contains the union of the elements residing in two filters by simply bitwise or'ing them (note: they must possess the same length

1. Burton Bloom, the original creator of the Bloom filter and author of [4], did not refer to this data structure as the "Bloom filter". It was simply referred to as "Method 2". However, it is now ubiquitously called the "Bloom filter" in reference to the creator.

and hash operations). Finally, as one can only insert and search for an element in Bloom filters, an inserted element is always guaranteed to be found (no false negatives).

All of the Bloom filter's properties, most notably the constant filter size, motivate the feasibility of tracking unique tainted flows through Bloom filters. The basic idea of this technique is to assign every register r a filter that keeps track of the unique taints that have propagated to r. If a new taint is generated at a register, then that taint is added as a unique element to the register's filter. The Bloom filter propagation logic mirrors the timing flow tracking logic. Note that we only track taint at registers (and monitored labels) because timing flows can only be blocked and generated at registers. Timing flows will propagate without any blockage over combinational logic. We now describe the technique in detail.

9.1 Description

The Bloom Filter Approach first executes the Taint Generation Logging Technique to determine the set of registers, R', that generate tainted flow during the faulty simulations. We then utilize the size of R' and a maximal false positive rate of 5% (arbitrary rate from our own choosing) to determine the appropriate Bloom filter size N (with upper bound of 256 bits) and number of hash functions d. Note that a Bloom filter with 256 bits and 4 hash functions can carry about 41 elements before its expected false positive rate exceeds 5% (calculated using [12]). Every register r that affects the monitored labels is given a Bloom filter label of size N called r_bloom_filter. Then for each register $r' \in R'$, we generate a random ID number and then hash it d times to determine the location of r's bits in the filter. From these d locations, we generate a $bloom\_filter\_id$ of size N for r'. For each of r''s procedural assignments $r' <= u$, this technique adds the following logic to the design:

$$r'\_bloom\_filter <= ((\text{all controllers are sensitive or not fully controlling})\ ? \tag{9.1}$$
$$(u'_0\_bloom\_filter \mid u'_1\_bloom\_filter \mid ... \mid u'_n\_bloom\_filter) : 0) \mid \tag{9.2}$$
$$(s_{r'_0}\_bloom\_filter \mid s_{r'_1}\_bloom\_filter \mid ... \mid s_{r'_m}\_bloom\_filter) \mid \tag{9.3}$$
$$((!is\_bal(r')\ \&\ (\text{one of } (r' <= u)\text{'s control signals carries tainted functional flow}))\ ? \tag{9.4}$$
$$(r'\_bloom\_filter\_id) : 0) \tag{9.5}$$

Where:

- $U = \{u'_0, u'_1, ..., u_n\}$ is the set of registers that are adjacent to and affect ("drive") u if u is not a register. If u is a register, then $u'_0 = u$ and $n = 0$.
- $\{u'_0\_bloom\_filter, u'_0\_bloom\_filter, ..., u'_n\_bloom\_filter\}$ is the set of corresponding Bloom filter labels for each register in U.
- $S_{r'} = \{s_{r'_0}, s_{r'_1}, ..., s_{r'_m}\}$ the set of registers that drive the control signals of $r' <= u$.
- Correspondingly, $\{s_{r'_0}\_bloom\_filter, s_{r'_1}\_bloom\_filter, ..., s_{r'_m}\_bloom\_filter\}$ is the set of Bloom filter labels for each register in $S_{r'}$.

The added logic is similar to the timing flow tracking logic from registers. The bitwise or's found in lines 9.1 to 9.3 merge the driving registers' Bloom filters into one Bloom filter that contains all of the unique taints that affect r''s timing label. Note that line 9.1 merges the Bloom filters affecting u only when the tainted flow propagating from u is not blocked (thus handling the blockage case). When a new taint is generated at r' at line 9.4, line 9.5 adds to the Bloom filter a new ID corresponding to that taint. Similarly, for every other register

x in the design, the approach adds the following logic for each assignment $x <= u$:

$$x\_bloom\_filter <= ((\text{all controllers are sensitive or not fully controlling})\ ? \tag{9.6}$$

$$(u'_0\_bloom\_filter \mid u'_1\_bloom\_filter \mid ... \mid u'_n\_bloom\_filter) : 0) \mid \tag{9.7}$$

$$(s_{x_0}\_bloom\_filter \mid s_{x_1}\_bloom\_filter \mid ... \mid s_{x_m}\_bloom\_filter) \tag{9.8}$$

This logic does not include a fourth or fifth line because we know that x does not generate tainted timing flow. Finally, for each monitored label l, an N-bit sized Bloom filter called l_bloom_filter is added to the design and the following logic is added (if the label is not a register's label):

$$l\_bloom\_filter <= d_0\_bloom\_filter \mid d_1\_bloom\_filter \mid d_k\_bloom\_filter \tag{9.9}$$

where $\{d_0\_bloom\_filter,\ d_1\_bloom\_filter, ..., d_k\_bloom\_filter\}$ is the set of Bloom filters of the registers that drive the value of the monitored label. This logic uses bitwise or's to merge all driving registers' Bloom filters. This merge captures all unique tainted flows coming into the monitored label from every adjacent register.

Once all of the logic has been added, the technique then simulates the design on the faulty simulations. It keeps each simulation's monitored labels' Bloom filters (note that it does not combine the filters). Finally, for each register in the candidate set proposed by the Taint Generation Logging Technique, check if it is in one of the simulations' Bloom filters. If it is, then add it to a set C_{bf}. Once each register has been checked, then C_{bf} is proposed as the bug location.

9.2 Accuracy Evaluation

In this section, we provide a qualitative evaluation of the Bloom Filter Approach's accuracy. We also describe how this technique can achieve 100% accuracy on the false positive examples found in Chapters 5 and 7.

As a search over a Bloom Filter can never result in a false negative, at worst, the Bloom Filter Approach has the same accuracy as the Taint Generation Logging Technique and less accuracy than the AGW Technique. That is, it is possible for all of the taint generating registers, even the false positives, to be found in the monitored label's Bloom filter. However, as long as the number of taint generating registers is not huge (e.g. not more than 42), the Bloom filter should have a false positive rate of 5% or less and thus the approach should identify very few false positives. This false positive rarity should make this technique much more accurate than both Taint Generation Logging and AGW Techniques. Note that since we limit each filter's size to a maximum of 256 bits, if a design has a large number of taint generating registers, the false positive rate has to increase, which leads to less accuracy. However, even if the number of taint generating registers is large like 100, the false positive rate is less than 30% which can still result in better accuracy than the AGW Technique's (calculated using [12]). For example, the Bloom Filter Approach would perform better if there exists many disjointed registers whose generated taint propagates to registers adjacent to or on an insecure path. We now describe how this technique would theoretically fare under the Taint Generation Logging Technique's and AGW Technique's false positive examples, both empirical and handmade, previously identified in Chapters 5 and 7. For the sake of simplicity, let the Bloom filters carry 8 bits and there only be one hash function.

Consider the example provided in Figure 5.1. Let $r1$'s Bloom filter ID be 00000001 and $r6$'s be 00100000. Then, during the simulation $r7$'s Bloom filter would become 00100000. $r2, r3$, and $r5$'s Bloom filters will be 00000000. The monitored label will therefore have a Bloom filter of 00100000. When the technique checks the monitored label's Bloom filter against the set of taint generating registers, it accurately produces $r6$ as the bug location.

Now consider the example in Figure 5.2. Let $r1$'s Bloom filter ID be 00000001 and $r6$'s be 00100000. During the first simulation, all other registers' Bloom filters will be 00000000. In the second simulation, $r3$'s Bloom filter becomes 00000001 but every other register's Bloom filter remains the same value. In both simulations, the monitored label will therefore possess the Bloom filter 00100000. When the technique checks the monitored label's Bloom filters against the set of taint generating registers, it accurately produces $r6$ as the bug location.

The example described in Chapter 5.3.5 is an example with a blockage point. Let $aes\_128.a2.qdiv\_fix\_ift.reg\_working\_dividend$'s filter ID be 00000001, $aes\_128.a2.qdiv\_fix\_ift.reg\_working\_quotient$'s be 00000010, $aes\_128.a5.qdiv\_fix\_ift.reg\_working\_dividend$'s be 00000100, and $aes\_128.a5.qdiv\_fix\_ift.reg\_working\_quotient$'s be 00001000. During the simulation, the Bloom filters from the latter two registers will propagate to the monitored label while the first two's filters will be blocked at the register $aes\_128.a2.qdiv\_fix\_ift.reg\_quotient\_fix$. The monitored label will therefore possess the Bloom filter 00001100. When the technique checks the monitored label's Bloom filter against the set of taint generating registers, it accurately chooses $aes\_128.a5.qdiv\_fix\_ift.reg\_working\_quotient$ and $aes\_128.a5.qdiv\_fix\_ift.reg\_working\_dividend$ as the bug location.

We now analyze the example in Figure 7.7. Let $r1$'s Bloom filter ID be 00000001 and $r3$'s be 00000100. During the simulation, all other registers' Bloom filters will be 00000000. Moreover, $r1$'s filter does not propagate to $r3$'s filter because of the multiplexer's input choice. The monitored label will therefore possess the Bloom filter 00000100. When the technique checks the monitored label's Bloom filter against the set of taint generating registers, it accurately identifies $r3$ as the bug location.

Finally, the example described in Figure 7.9 is an example with a blockage point that causes false positives in the AGW Technique. Let $r1$'s Bloom filter ID be 00000001, $r2$'s be 00000010, and $r3$'s be 00000100. During the first simulation, the monitored label's Bloom filter will be 00000100 since $r1$'s taint is blocked and $r2$ does not generate tainted flow. In the second simulation, the monitored label's filter is 00000110 since $r2$ now generates tainted flow. Every other register's Bloom filter remains the same value. When the technique checks the monitored label's Bloom filters against the set of taint generating registers, it accurately produces $r2$ and $r3$ as the bug location.

Thus, through our reasoning and evaluation on the handmade examples, the Bloom Filter Approach can greatly reduce the incidence of false positive registers.

9.3 Efficiency Evaluation

The Bloom Filter Approach is more computationally efficient than most techniques. It utilizes relatively little memory and requires relatively little time. For this (informal) efficiency evaluation, we define the following variables:

- m = the number of monitored labels.
- s = number of simulations.
- r = number of registers that affect the monitored label.
- c = maximum number of cycles in a simulation.

We define the following costs for our complexity analysis:

- $O(n)$ = the worst-case cost of executing all of a simulation's operations (e.g. register and combinational logic assignments). This number also includes the cost of executing the IFT logic.
- $O(1)$ = the time cost of querying a Bloom filter for an element.
- $O(x)$ = the maximal memory footprint of an unmodified simulation.

Furthermore, for simplicity, we assume that the action of logging the simulations' behavior (registers, IFT, control signals) takes $O(1)$ time and space.

9.3.1 Time Complexity

During simulation, the Bloom filter propagation and insertion logic are very computationally lightweight as they mostly consist of bitwise or's. These operations therefore should not introduce overhead that eclipses the overhead of the Taint Generation Logging Technique. More specifically, they should possess about the same runtime as normal timing flow tracking logic. The Bloom Filter approach introduces at most, one additional flow tracking assignment for each register assignment. The worst-case cost of executing a simulation with Bloom filters is thus $O(\text{constant} * n) = O(n)$. The worst-case for all simulations is therefore $O(sn)$. The time complexity of localization will be $O(usm)$ because all the technique has to do is query each monitored label's Bloom filter over all simulations for each unbalanced register. Also note that an additional $O(src)$ time is required in order to run the Taint Generation Logging Technique to first determine the taint generating registers. Thus, the time complexity is $O(sn) + O(src) + O(usm) = O(s(n + um + rc))$

9.3.2 Space Complexity

The memory overhead during simulation is $O(x)$ because the size of each filter and filter ID will be at most 256 bits, meaning that the total added memory is less than or equal to some constant times the number of registers (which is less than some constant times x). The localization portion requires $O(\text{constant} * ms + src)$ space since it runs the Taint Generation Logging Technique's localization step and also utilizes the monitored labels' filters from each simulation. The overall space complexity is therefore $O(x + ms + src)$.

CHAPTER 10
PROBABILISTIC PATHING

We observed in previous chapters that tainted flow can be stopped from propagating due to control flow behavior at multiplexers or timing flow blockage points. We make a second observation that if a tainted flow is often or always stopped, then the probability of it reaching the monitored labels over all simulations should be low. The Probabilistic Pathing Technique therefore utilizes this new observation to determine the bug generating location. It utilizes control signal traces to compute a score that reflects the chance that a generated taint eventually affects the monitored label. Note that this score is not a probability (though it is the sum of probabilities). Also note that technique does not eliminate all false positives, but it allows for the designer to make an well-informed decision as to whether a register is likely a false positive. We now describe the technique in detail.

10.1 Description

This technique first executes a modified Taint Generation Logging Technique. The Taint Generation Logging Technique now also tracks the values of all control signals in a control signal matrix (similar in structure to the original trace matrices). In addition, the Taint Generation Logging Technique tracks the values of all timing flow blockage logic in the design. The technique considers each timing flow blockage point as a multiplexer with two inputs:

1. flow of the right-hand side signals | timing flow propagation from control signals | timing flow generation result
2. timing flow propagation from control signals | timing flow generation result

The control signal for that "multiplexer" is the value of the timing flow blockage logic. If flows are not blocked, then the first input is chosen; otherwise, the second input is chosen.

Now for each multiplexer M and each of its inputs i_M, we utilize the control signal traces to determine the average proportion of a simulation, i.e. probability, that i_M is chosen at M. Call this probability pr_{i_M}. Then for each register r in the candidate set produced by the Taint Generation Logging Technique, the technique:

1. Determines all paths from r to the monitored labels. As an optimization step, if a path has a data loop, only traverse that loop once. Call this set of paths P.

2. For each path $p \in P$, let $\{M_0, ..., M_n\}$ be the set of multiplexers along the path. Utilize the set of input probabilities at each multiplexer to calculate the probability of that path. That is, let $i_{M_0}, i_{M_1}, ..., i_{M_n}$ be the set of inputs at the multiplexer that constructs the path. Then,

$$p\_path\_probability = pr_{i_{M_0}} pr_{i_{M_1}} ... pr_{i_{M_n}}$$

3. Sum the path probabilities for all $p \in P$ to get the score for r. Again, this score is a sum of probabilities and not an actual probability itself. However, even though it is not a probability, it tells the programmer how likely a candidate register will propagate its flow to the monitored label.

Once all registers in the candidate set have been scored, the technique orders the registers by score and then presents them to the programmer. It is then up to the designer to decide whether a register is a false positive or part of the bug location, e.g. through a scoring threshold.

10.2 Accuracy Evaluation

In this section, we provide a qualitative evaluation of the Probabilistic Pathing Technique's accuracy. We describe how the programmer can analyze this technique's results to achieve 100% accuracy on the AGW Technique's and Taint Generation Logging Technique's false positive examples. Note that this technique does not explicitly remove the disjointed register false positives. However, with careful analysis on the technique's output, most, if not all, false positives can be identified and removed.

Now consider the example provided in Figure 5.1. The probability of the only path from $r1$ to the monitored label, $r1 \rightarrow r3 \rightarrow \textit{monitored\_label}$ is 0 because the input chosen at the top multiplexer is always the $r2$ input. Thus, the score is 0. The register $r6$ is always chosen at the lower multiplexer, meaning that the path $r6 \rightarrow r7 \rightarrow \textit{monitored\_label}$ will have a probability of 1. Thus, the designer will be provided with the following ranking:

1. $r6 = 1$
2. $r1 = 0$

The programmer can then infer that $r6$'s taint has a very high chance of flowing to the monitored label while $r1$'s taint has no chance. Thus, the designer can accurately determine that the register $r6$ is the bug location.

Figure 5.2's example is slightly different from Figure 5.1's example because the false positive register $r1$ attains a non-zero score. The probability of the only path from $r1$ to the monitored label, $r1 \rightarrow r3 \rightarrow r5 \rightarrow \textit{monitored\_label}$ is $1/4$ because the chance of $r1$ being chosen at the leftmost multiplexer is $1/2$ and the chance of $r3$ being chosen at the rightmost multiplexer is $1/2$. Thus, the score is $1/4$. The register $r6$ does not have a multiplexer in front of it and thus $r6$ will always have a score of 1. The score ranking would therefore be:

1. $r6 = 1$
2. $r1 = 1/4$

The programmer can then deduce that $r6$'s taint has a very high chance of flowing to the monitored label. Because $r1$'s score is non-zero, the programmer cannot say for certain that $r1$'s taint will not propagate to the monitored label. The designer, however, can look at $r1$'s low score and infer that its taint's propagation to the monitored label is unlikely. The designer would therefore accurately identify only the register $r6$ as the bug location.

The example described Chapter 5.3.5 is a real example with a blockage point. As we have not implemented this technique yet, we do not have the register scores. However, we can and will describe the general behavior of the technique when it is applied to this example. The blockage point at $aes\_128.a2.qdiv\_fix\_ift.reg\_quotient\_fix$ will cause the scores of $aes\_128.a2.qdiv\_fix\_ift.reg\_working\_dividend$ and $aes\_128.a2.qdiv\_fix\_ift.reg\_working\_quotient$ to be 0. On the other hand, the scores of $aes\_128.a5.qdiv\_fix\_ift.reg\_working\_dividend$ and $aes\_128.a5.qdiv\_fix\_ift.reg\_working\_quotient$ will be non-zero. The designer would therefore accurately choose $aes\_128.a5.qdiv\_fix\_ift.reg\_working\_dividend$ and $aes\_128.a5.qdiv\_fix\_ift.reg\_working\_quotient$ as the bug location.

We now consider the example in Figure 7.7. The probability of the only path from $r1$ to the monitored label, $r1 \rightarrow r3 \rightarrow monitored\_label$ is 0 because the chance of $r1$ being chosen at the multiplexer is 0. Thus, the score is 0. The register $r3$ does not have a multiplexer in front of it and thus it will always have a score of 1. The score ranking would therefore be:

1. $r3 = 1$

2. $r1 = 0$

The programmer can identify that $r3$'s taint is likely to propagate to the monitored label. Since $r1$'s score is zero, the programmer can definitively tell that $r1$'s taint will not propagate to the monitored label. The designer would therefore accurately identify only the register $r3$ as the bug location.

Finally, the example described in Figure 7.9 is an example with a blockage point that causes false positives in the AGW Technique. The only path from $r1$ to the monitored labels, $r1 \rightarrow r2 \rightarrow monitored\_label$, will have a score of 0 because it is always blocked from propagating. Both $r3$ and $r2$ will both have a score of 1 because there are no blockage points or multiplexers ahead of them. The score ranking would therefore be:

1. $r3 = 1$

2. $r2 = 1$

3. $r1 = 0$

The programmer would therefore identify that $r3$ and $r2$'s taints are likely to propagate to the monitored label. Since $r1$'s score is zero, the programmer can tell that $r1$'s taint will not propagate to the monitored label. The designer would therefore accurately determine that the registers $r2$ and $r3$ are the bug location.

10.3 Efficiency Evaluation

Probabilistic Pathing's computational complexity is similar to the AGW technique's. We define the following variables for our worst-case complexity analyses:

- s = number of simulations.
- r = number of registers that affect the monitored label.
- u = number of unbalanced registers that affect the monitored label. Note that the number of taint generating registers is less than or equal to u.
- t = number of simulations.
- c = maximum number of cycles in a simulation.
- n = the maximum number of operations in an unmodified simulation.

We define the following costs for our complexity analysis:

- $O(1)$ = the worst-case time cost of executing one of a simulation's operations at a cycle (e.g. IFT logic, register assignments, and combinational logic assignments). Note then, that the worst-case time complexity of a simulation is $O(n)$ and the time complexity of running all simulations is $O(sn)$.
- $O(1)$ = the time it takes to traverse from a node to edge or edge to node.

- $O(1)$ = the time it takes to compute the score of a register once the paths have been found.
- $O(x)$ = the maximal memory footprint of an unmodified simulation.
- $O(1)$ = the memory footprint of a node or edge in a path.

Furthermore, for simplicity, we assume that the action of logging the simulations' behavior (registers, IFT, control signals) takes $O(1)$ time and space.

10.3.1 Time Complexity

The Probabilistic Pathing Technique does not modify the design and thus the simulation time is $O(sn)$. Assuming that this technique optimizes out loops, its localization time complexity will be $O(r^2)$ for each candidate register it considers. This is because it can run DFS or BFS search to find all of the paths between a candidate register and monitored label (which gives $O(r^2 + r) = O(r^2)$ complexity). Therefore, this technique's localization time complexity is $O(ur^2)+O(src)$ (because of the Taint Generation Logging Technique's localization portion). The total time complexity is therefore $O(s(n + rc) + ur^2)$.

10.3.2 Space Complexity

The memory footprint during simulation is $O(x)$ because the technique only modifies the design when it executes the Taint Generation Logging Technique. During localization, $O(src)$ space is required for the Taint Generation Logging Technique's localization portion. In addition, the path determination procedure's memory footprint is at worst $O(ur^2)$ because the technique has to store all registers and edges when it finds the paths between each candidate register and the monitored labels. Therefore, the memory complexity is $O(x + ur^2 + src)$.

Table 11.1: A summary of the accuracy and informal worst-case computational complexities of each technique.

Technique	Accuracy	Time Complexity	Space Complexity
Selective Taint Generation	100%	$O(u(k+sn))$	$O(x+sm)$
Taint Generation Logging Technique	<100% - many disjointed register false positives cases exist	$O(s(n+rc))$	$O(x+src)$
Brute-Force Backtracking	100%	$O(s(mb^c+n))$	$O(x+mb^c)$
Aggregated Graph Walking	<100% - some disjointed register false positives cases exist	$O(s(n+rc)+t^2)$	$O(x+t^2+src)$
Unique Taint Tracking	100%	$O(s(d+u^2m))$	$O(u(sm+x)+src)$
Bloom Filter Approach	$\approx$ 100% - if false positive rate is low (which it usually is)	$O(s(n+um+rc))$	$O(x+ms+src)$
Probabilistic Pathing	Variable, can help the designer to achieve 100% accuracy, but depends on designer's decisions	$O(s(n+rc)+ur^2)$	$O(x+ur^2+src)$

CHAPTER 11

SUMMARIES AND COMPARISONS OF TECHNIQUES

In this chapter, we summarize and then compare the techniques. Furthermore, we organize these summaries and comparisons into two tables (Table 11.1 and 11.2).

11.1 Technique Summaries

This section summarizes the main idea of and how we evaluated each technique.

1. Selective Taint Generation: Test whether an unbalanced register r's generated taint reaches the monitored label by turning off the tainted timing flow generation logic for

Table 11.2: A summary of the main strengths and weaknesses of each technique.

Technique	Strengths	Weaknesses
Selective Taint Generation	Achieves 100% accuracy and relatively simple to implement.	Space and time inefficient.
Taint Generation Logging	Efficient and relatively simple to implement.	Low accuracy; there exist many disjointed register false positives.
Brute-Force Backtracking	Achieves 100% accuracy.	Very space and time inefficient.
Aggregated Graph Walking	Achieves relatively high accuracy while also having relatively good efficiency.	Somewhat space and time inefficient. Still vulnerable to some disjointed register false positives.
Unique Taint Tracking	Achieves 100% accuracy	Very space and time inefficient.
Bloom Filter Approach	Can often achieve ≈ 100% accuracy and extremely space and time efficient.	If the number of buggy registers in a design is huge, then this approach may suffer from many false positives.
Probabilistic Pathing	Gives the designer the freedom to decide which register is or is not a buggy register. May allow the designer to determine the bug location with 100% accuracy.	Final buggy register determination is left to the designer, which may result in human-induced inaccuracy. Somewhat space and time inefficient.

all registers except r's and then running the simulations.

- Accuracy Evaluation: Straightforward reasoning about the techinque's properties.
- Efficiency Evaluation: Informal worst-case complexity analysis.

2. Taint Generation Logging Technique: Find the set of registers that generate tainted timing flow during faulty simulations. The bug location must be a subset of this set of registers.

   - Accuracy Evaluation: Tested the technique on real examples and reasoned about its accuracy on handmade/theoretical designs.

- Efficiency Evaluation: Informal worst-case complexity analysis.

3. Brute-Force Backtracking: For each faulty simulation, walk backwards from monitored labels to adjacent registers along insecure paths to the bug location.
    - Accuracy Evaluation: Straightforward reasoning about the techinque's properties.
    - Efficiency Evaluation: Informal worst-case complexity analysis.
4. Aggregated Graph Walking: Aggregate the IFT behavior of all faulty simulations and use a graph walking algorithm to generate a register dependency graph containing all insecure paths and buggy registers.
    - Accuracy Evaluation: Tested the technique on real examples and reasoned about its accuracy on handmade/theoretical designs.
    - Efficiency Evaluation: Informal worst-case complexity analysis.
5. Unique Taint Tracking: Track the propagation of tainted flows uniquely identified by their originating location with arrays at each register.
    - Accuracy Evaluation: Straightforward reasoning about the techinque's properties.
    - Efficiency Evaluation: Informal worst-case complexity analysis.
6. Bloom Filter Approach: Utilize Bloom filters to efficiently track unique tainted timing flows.
    - Accuracy Evaluation: Reasoned about the technique's accuracy on handmade/theoretical designs.
    - Efficiency Evaluation: Informal worst-case complexity analysis.
7. Probabilistic Pathing: Compute the chance of a tainted timing flow propagating from a register to the monitored labels.

- Accuracy Evaluation: Reasoned about the technique's accuracy on handmade/theoretical designs.
- Efficiency Evaluation: Informal worst-case complexity analysis.

11.2 Technique Analyses and Comparisons

Tables 11.1 and 11.2 summarize each technique's accuracy, (informal) computational complexities, strengths, and weaknesses. The following is a summary of the variables and runtime assumptions utilized in these tables:

- m = the number of monitored labels.
- s = number of simulations.
- u = number of unbalanced registers that affect the monitored labels. Note that the number of taint generating registers is less than or equal to u.
- r = number of registers that affect the monitored labels.
- t = number of taint carrying registers that affect the monitored labels.
- c = number of cycles during the simulation.
- b = the maximum branching factor of the "rolled out" data flow tree traversed by the DFS or BFS algorithm.
- n = the maximum number of operations during an unmodified simulation.

We define the following costs for our complexity analysis:

- $O(1)$ = the worst-case time cost of executing one of a simulation's operations at a cycle (e.g. IFT logic, register assignments, and combinational logic assignments). Note then, that the worst-case time complexity of a simulation is $O(n)$. Moreover, the worst-case time complexity of simply running all simulations is $O(sn)$.

- $O(1)$ = the time it takes to traverse from a node to edge or edge to node in the Probabilistic Pathing, AGW, and Brute-Force Backtracking Techniques.
- $O(1)$ = the time cost of querying a Bloom filter for an element.
- $O(1)$ = the time it takes to retrieve and compare an element in a taint array or trace matrix (Unique Taint Tracking or Taint Generation Logging Techniques).
- $O(1)$ = the time complexity of checking whether the monitored labels were violated during a simulation for the Selective Taint Generation Technique.
- $O(1)$ = the time it takes to compute the score of a register once the paths have been found (Probabilistic Pathing Technique).
- $O(k)$ = the time it takes to modify and recompile the design. We assume that $k >> 1$.
- $O(d)$ = the worst-case time cost of concatenating and deduplicating the arrays over all unique taint tracking operations. We assume that $d >> 1$ and $d >> n$.
- $O(x)$ = the maximal memory footprint of an unmodified simulation.
- $O(1)$ = the memory footprint of a monitored label during the localization part of the Selective Taint Generation technique.
- $O(1)$ = the memory footprint of a node or edge in a path, graph, or tree in the Probabilistic Pathing, AGW, and Brute-Force Backtracking Techniques.
- $O(1)$ = Each taint array's element's memory footprint in the Unique Taint Tracking Technique.

Generally, higher accuracy coincides with greater time complexities. The three most accurate techniques, Selective Taint Generation, Brute-Force Backtracking, and Unique Taint Tracking, respectively possess the time complexities: $O(u(k+sn))$, $O(s(mb^c+n))$, and $O(s(d+u^2m))$. The first complexity is very large because $k >> 1$, $sn >> 1$, and u is often greater than 1, which implies that $u(k + sn)$ will be a very large value. The second complexity is also very large because its value is exponential and multiplied by the number of simulations and monitored labels. Finally, the third complexity is large because $d >> n$. The value d is much greater than n because $O(d)$ encodes the huge amount of resources required to dynamically concatenate and then deduplicate a multitude of large arrays for each register assignment.

As the techniques become less accurate, they generally become more time-efficient. The Taint Generation Logging Technique, which is likely to possess less than 100% accuracy, has an $O(s(n + rc))$ time complexity. We observe that $rc < n$ since n accounts for **all** operations during a simulation, including the operations at the r registers for each of the simulation's c cycles. Thus, $O(sn) > O(src) \implies O(s(n + rc)) = O(sn)$, which is much more efficient that the perfectly accurate techniques. The AGW Technique's time complexity is $O(s(n + rc) + t^2) = O(sn + t^2)$. We assume that t^2 is usually not greater than (or at least is not much greater than) sn. Our justification for this assumption is that often, $t << n << sn$ and thus $t^2 \leq n \leq sn$. We now explain why this justification is true. First observe that $t \leq r \leq rc \leq n \leq sn$. In practice, simulations often run for at least a few thousand cycles, meaning that c is often very large $\implies$ usually, $r << rc$. r is also likely to be greater than t because there is a significant chance that some registers in a design will never carry tainted flow (e.g. registers that are positioned before the buggy registers). In addition, in most cases, the number of simulations is much greater than 1 ($s >> 1$). Finally, designs usually possess many more wire operations (i.e. combinational logic operations) than register operations, meaning that rc is often much less than n. These insights together imply that usually, $t << rc << n << sn \implies t << n << sn \implies t^2 \leq n \leq sn$. Based on this assumption, we can therefore infer that the AGW technique is usually only slightly to moderately more inefficient than the Taint Generation Logging Technique. Using this inference, we can further deduce that AGW technique is very likely to be more efficient than the three techniques discussed above. Finally, using similar reasoning, we infer that

Probabilistic Pathing has a slightly higher time complexity than the AGW Techique's that is still less than the three perfectly accurate techniques' complexity.

A notable exception to the trends we identified above is the Bloom Filter Approach. Its time complexity is relatively small: $O(s(n+um+rc))) \approx O(sn+sn) = O(sn)$ since usually $m << c \implies$ usually $um < rc < n$. It also has the potential to achieve close to 100% accuracy. Thus, the Bloom Filter Approach appears to be the most promising approach to study in the future. We will discuss future works involving this approach in the subsequent chapter.

Most of the techniques' space complexities (found in Table 11.1) are reasonable and do not blow up. That is, their complexities only require some extra memory added on top of the original maximal memory footprint $O(x)$. However, the Brute-Force Backtracking and Unique Taint Tracking Techniques' space complexities are very large ($O(x + mb^c)$ and $O(u(sm + x) + src)$ respectively). The former technique's memory footprint is large as it is tied to the exponential number of nodes in the "rolled" out trees. The latter technique's complexity is large because it is multiplies the maximal memory footprint x by u, which could have serious repercussions even when u is relatively small. E.g. 20 unbalanced registers means 20 times the memory usage. Therefore, the application of these two techniques may not be feasible on larger, longer running, and more complex designs.

CHAPTER 12
OTHER DISCUSSION

In this chapter, we discuss some other insights we gained about the techniques.

12.1 Optimal Number of Faulty Simulations

We utilized varying amounts of faulty simulations during empirical experiments. For example, the AES with divider experiments possessed around 80 to 90 faulty inputs. While the input sets we utilized were large enough to expose all registers that comprise the example designs' bug locations, the same number of inputs may not be enough for other, more complex designs. In the ideal case, the techniques should simulate the design on all possible faulty inputs to capture all possible insecure paths and taint generating registers. Using all possible inputs, however, is not practical because a design can possess an infinite amount of inputs. A more realistic approach would be to utilize at least one input from each input class. An input class is a set of inputs whose simulations' timing label behavior is the same. That is, the same number of cycles executed, time and location of tainted timing labels, monitored label values, etc. However, identifying all input classes is a difficult task if there exists a substantial number of input classes. Utilizing a large number of random inputs, which requires relatively little effort to identify, may therefore be the most practical approach for now.

12.2 Using the Techniques to Fix Bugs

Most of the techniques, e.g. the Taint Generation Logging and Brute-Force Backtracking Techniques, provide the designer with either a candidate set or the actual set of buggy registers. Moreover, their candidate sets will always contain the bug generating registers; they will never leave out buggy registers. The designer can therefore fix the timing-based leakage by implementing non-sensitive and fully controlling signals, e.g. through counters, for the flows at those registers.

Techniques like the AGW Technique, however, provide the designer with better options for fixing timing-related security bugs. While we define the bug location to be all of the registers that generate monitored label-affecting tainted flow, fixing the bug may be as simple as implementing logic to block timing flows at or near just a few registers. For example, the dependency graph in the AES with divider examples indicates that simply blocking the tainted flow at or near *aes_128.a5.qdiv_orig_ift.reg_working_quotient* with a counter is enough to stop the security invariants from being violated. That is, *aes_128.a5.qdiv_orig_ift.reg_working_quotient* is a tainted timing flow "choke point". Another example is the cache, where simply blocking timing flows at *Cache.Mem32.mem* with a counter is enough to stop the monitored label from getting tainted. Dependency graphs are therefore important for bug patching because the expose these "choke points".

12.3 Technique Automation

While we didn't implement many of the techniques, the strategies can easily be automated. We have already defined the techniques' procedures and thus their implementations are simply a software exercise.

CHAPTER 13
FUTURE WORKS AND CONCLUSION

13.1 Future Works

13.1.1 Implementation and Empirical Evaluation of the Techniques

We conducted an empirical analysis on only two of the seven techniques. In addition, the empirical evaluations on these two techniques were for accuracy and not efficiency. The other five techniques were also only evaluated informally and qualitatively. Future works should therefore implement all seven techniques and evaluate **both** their accuracies and efficiencies on a multitude of designs. Moreover, as the Bloom Filter Approach appears to be the most promising technique (efficient and highly accurate), future works should apply extra focus to its evaluation.

13.1.2 Complexity and Size of Empirically Tested Designs

The example designs we tested in this paper were relatively simple and small, with the most complex hardware being the cache and RSA designs. As we discovered, this study's examples were too simple to reveal common false positive cases in the Taint Generation Logging and AGW Techniques. The only case that revealed false positives was the AES with divider and timing flow blockage example. The longest running design also only ran on the order of tens of thousands of cycles. Future works should therefore experiment the techniques on a multitude of larger and more complex designs. Moreover, many of the new examples should be able to produce the false positives found through the handmade examples. The application of the techniques to more complex designs should also test for both their localization efficiency and accuracy.

13.1.3 Technique Optimizations

We determined in Chapter 11 that the Probabilistic Pathing Technique possesses more time complexity than the AGW Technique. Future works should therefore focus on optimizing the Probabilistic Pathing Technique's efficiency. One promising approach involves only choosing the taint generating registers along n high probability paths as the bug location. The optimized technique finds common/very probable flow propagation paths by backtracking from the monitored labels. Whenever the technique encounters a multiplexer, it uses its control signal trace to choose an input that has high probability of being chosen. After finding the n paths, the technique then determines all taint generating registers along those paths. Those registers are proposed as the bug location. This optimized technique is much more efficient than the Probabilistic Pathing Technique; it only makes a constant number of backtracks. At the same time, this technique runs the risk of leaving bug location registers out of its candidate set since it only analyzes a constant number of paths. Future works should therefore also conduct extensive efficiency and accuracy evaluations on this optimized technique with a multitude of complex designs.

Future works should also explore whether the three perfectly accurate techniques can be optimized without any accuracy compromises. If such optimizations can be identified, then we would have very powerful techniques that could be effectively utilized in hardware verification and synthesis techniques like VeriSketch.

13.1.4 Non-Timing Security Bugs (Functional Flows)

As not all security bugs are timing-related, future works should examine how to localize security bugs in general. That is, bugs related to tainted functional flows, tainted timing flows, or a mixture of tainted functional and timing flows. The first steps toward general localization would therefore be to develop techniques to localize functional flow-related bugs.

Functional flow possesses a few differences from timing flows. The first difference is that tainted functional flows cannot be generated within a hardware design. They can only be propagated from input to output. Thus, the definition of a functional flow bug must actually be the insecure paths from input to output so that the programmer knows where to block the tainted timing flow. Second, tainted functional flow can be blocked in combinational logic as well.

The Bloom Filter and Unique Taint Tracking Techniques will not work with functional flow-related bugs because they are specifically built for tracking unique tainted flow generation. However, other techniques like the AGW and Probabilistic Pathing Techniques may be generalizable to functional flows. For example, the AGW's graph generation phase produces the insecure paths, and thus the dependency graph would serve as the bug location of functional flows. The Probabilistic Pathing technique can calculate the probability for some taint to propagate on a path to the monitored labels. This technique could thus help the programmer to localize functional flow-related bugs by calculating the probability of a functional taint-carrying path that ends at the monitored labels. Future works should therefore look into modifying our strategies to allow them to localize functional flow-related bugs.

13.2 Conclusion

In this paper, we claimed that IFT can be utilized to automatically and accurately localize timing-related security bugs. We demonstrated our claim by proposing seven automated IFT-based timing bug localization techniques. We evaluated each of them through varying methods ranging from empirical experiments to informal complexity analyses. We found that these seven techniques generally had good accuracy and feasible efficiency. However, most of the evaluations were qualitative and thus, in future works, we intend to implement and evaluate them on a multitude of more insightful examples. In the future, we also hope to see work in improving our techniques' accuracy, efficiency, and generalizability to all security-related bugs. This work is the first of its kind to explore using IFT to localize timing-related security bugs. It is the first step in the construction of a larger IFT-based bug localization ecosystem that we hope will be heavily explored in the future.

REFERENCES

[1] A. Ardeshiricham, W. Hu, and R. Kastner. Clepsydra: Modeling timing flows in hardware designs. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 147–154, Nov 2017. doi: 10.1109/ICCAD.2017.8203772.

[2] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. Register transfer level information flow tracking for provably secure hardware design. In *Proceedings of the Conference on Design, Automation Test in Europe*, DATE '17, page 1695–1700, Leuven, BEL, 2017. European Design and Automation Association.

[3] A. Ardeshiricham, Y. Takashima, S. Gao, and R. Kastner. Verisketch: Synthesizing secure hardware designs with timing-sensitive information flow properties. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1623–1638, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3354246. URL `https://doi.org/10.1145/3319535.3354246`.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[5] T. Burke. Fixed point math library for verilog, Jan 2014. URL `https://opencores.org/projects/verilog_fixed_point_math_library`.

[6] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, Vancouver, BC, Aug. 2017. USENIX Association. ISBN 978-1-931971-40-9. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen`.

[7] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. Prodact: Prefetch-obfuscator to defend against cache timing channels. *International Journal*

of Parallel Programming, 47(4):571–594, Aug 2019. ISSN 1573-7640. doi: 10.1007/s10766-018-0609-3. URL `https://doi.org/10.1007/s10766-018-0609-3`.

[8] D. Gruss, C. Maurice, and K. Wagner. Flush+flush: A stealthier last-level cache attack. *CoRR*, abs/1511.04594, 2015. URL `http://arxiv.org/abs/1511.04594`.

[9] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., Aug. 2015. USENIX Association. ISBN 978-1-939133-11-3. URL `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss`.

[10] S. Gueron. Intel® advanced encryption standard (aes) new instructions set. Technical report, Intel Corporation, May 2010.

[11] A. Hagberg, D. Schult, and M. Renieris. Pygraphviz. URL `http://pygraphviz.github.io/`.

[12] T. Hurst. Bloom filter calculator, Oct 2018. URL `https://hur.st/bloomfilter/`.

[13] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 591–604, Los Alamitos, CA, USA, may 2015. IEEE Computer Society. doi: 10.1109/SP.2015.42. URL `https://doi.ieeecomputersociety.org/10.1109/SP.2015.42`.

[14] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017. doi: 10.475/1234.

[15] 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.

[16] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 109–120, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993512. URL `https://doi.org/10.1145/1993498.1993512`.

[17] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A language for hardware-level security policy enforcement. *SIGARCH Comput. Archit. News*, 42(1):97–112, Feb. 2014. ISSN 0163-5964. doi: 10.1145/2654822.2541947. URL `https://doi.org/10.1145/2654822.2541947`.

[18] 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)*, pages 973–990, Baltimore, MD, Aug. 2018. USENIX Association. ISBN 978-1-939133-04-5. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/lipp`.

[19] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss. Take a way: Exploring the security implications of amd's cache way predictors. In *15th ACM Asia Conference on Computer and Communications Security*, June 2020. doi: https://doi.org/10.1145/3320269.3384746.

[20] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015. doi: 10.1109/SP.2015.43.

[21] F. Liu, H. Wu, K. Mai, and R. B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, Sep. 2016. ISSN 1937-4143. doi: 10.1109/MM.2016.85.

[22] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129, June 2012. doi: 10.1109/ISCA.2012.6237011.

[23] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner. Leveraging gate-level properties to identify hardware timing channels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(9):1288–1301, 2014.

[24] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[25] S. Park and S. Mitra. Ifra: Instruction footprint recording and analysis for post-silicon bug localization in processors. In *2008 45th ACM/IEEE Design Automation Conference*, pages 373–378, June 2008.

[26] C. Percival. Cache missing for fun and profit. 08 2009.

[27] M. K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787, Oct 2018. doi: 10.1109/MICRO.2018.00068.

[28] H. Salmani and M. Tehranipoor. Chip-level trojan benchmarks. URL `https://www.trust-hub.org/benchmarks/chip-level-trojan`.

[29] H. Salmani, M. Tehranipoor, and R. Karri. On design vulnerability analysis and trust benchmarks development. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 471–474, 2013.

[30] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor. Benchmarking of hardware trojans and maliciously affected circuits. *Journal of Hardware and Systems Security*, 1, 04 2017. doi: 10.1007/s41635-017-0001-6.

[31] S. Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing, Apr 2015. doi: 10.1007/978-3-319-16214-0_42. URL `http://dx.doi.org/10.1007/978-3-319-16214-0_42`.

[32] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 109–120, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584065. doi: 10.1145/1508244.1508258. URL `https://doi.org/10.1145/1508244.1508258`.

[33] P. Vila, B. Köpf, and J. F. Morales. Theory and practice of finding eviction sets. *CoRR*, abs/1810.01497, 2018. URL `http://arxiv.org/abs/1810.01497`.

[34] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 494–505, New York, NY, USA, 2007. Association

for Computing Machinery. ISBN 9781595937063. doi: 10.1145/1250662.1250723. URL `https://doi.org/10.1145/1250662.1250723`.

[35] S. Williams. Icarus verilog, 2015. URL `http://iverilog.icarus.com/home`.

[36] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360, June 2017. doi: 10.1145/3079856.3080222.

[37] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, volume 1, pages 56–72, Los Alamitos, CA, USA, may 2019. IEEE Computer Society. doi: 10.1109/SP.2019.00004. URL `https://doi.ieeecomputersociety.org/10.1109/SP.2019.00004`.

[38] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 719–732, USA, 2014. USENIX Association. ISBN 9781931971157.

[39] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 503–516, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450328357. doi: 10.1145/2694344.2694372. URL `https://doi.org/10.1145/2694344.2694372`.

[40] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page

815–827. IEEE Press, 2018. ISBN 9781538662403. doi: 10.1109/MICRO.2018.00071. URL `https://doi.org/10.1109/MICRO.2018.00071`.