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

Dedication Text

Epigraph Text

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

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 first implement two techniques that can be utilized to automatically localize timing-related security bugs in hardware designs: a simple and a novel dependency graph walking technique. The main idea of these techniques involves the aggregation of timing behavior in fault-causing simulations. We can obtain the information needed to capture such timing variations by leveraging existing IFT infrastructure with low cost and high precision. We experiment these techniques on various hardware designs and evaluate their strengths and limitations. Based on these insights, we propose new techniques that can overcome many of the old techniques' limitations. We provide a theoretical analysis on their localization accuracy and efficiency. We leave the further elaboration, implementation, extension, and empirical evaluation of these new 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 word 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 extremely 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 cache side-channel exploits. These attacks exploit timing differences in cache accesses to infer secret information about a victim process, e.g. a cryptographic algorithm's secret keys. Cache attacks are especially dangerous since they are clever, lightweight, simple, and efficient. For example, some techniques can take advantage of cache directories while others can exploit transactional aborts to quickly leak data [32, 5]. Not only can cache attacks be utilized standalone, but also to augment other attacks. Out-of-order and speculative execution exploits such as Meltdown and Spectre [17, 14] use cache exploits to construct covert channels that can leak data to an attacking process. It is therefore imperative that exploit mitigations be implemented in all caches.

As with cache attacks, the hardware-based mitigations for such attacks come in a multitude of forms. The Partition-Locked Cache (PLCache) [30], for example, mitigates side-channels by allowing processes to dynamically partition the cache into private 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 the more complex ones. Without these tools, designers waste resources and may even unknowingly make mistakes during the design or implementation phase. For example, some researchers [3] recently constructed and formally analyzed a PLCache design. During their analysis, they found a subtle side-channel in which a specific set of accesses to the same cache set as a locked line can take advantage of the cache's replacement policy to induce timing-related data leakage. 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 [28, 2, 1, 3], e.g. gate level or Register-Transfer Level (RTL). The tracked data always possesses at least one label that denotes whether it contains sensitive information. Some strategies, such as Clepsydra and VeriSketch [1, 3], have developed IFT rules that separate information flows into timing-based flows (whether computation of some data contains timing variations), and functional flows [22] (whether the values of some data depends on the values of other data). As a result, the data in these techniques will possess two labels: one for the functional flow and one for the timing flow. However, while IFT techniques can be used to automatically detect security flaws, they have not been used to automatically find the origin of, or localize, such flaws. The hardware designer, who wishes to fix the bug, is therefore left with the arduous and not necessarily guaranteed task of finding the the bug's location.

We therefore focus on augmenting IFT strategies with techniques that can automatically

localize a timing-based security bug in any faulty hardware design. We specifically seek to utilize VeriSketch's and Clepsydra's IFT strategy to automatically identify the location of a detected bug [3, 1]. In this paper, we construct and explore two strategies 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 security related bugs. We explore this strategy's effectiveness on multiple buggy examples. Our results indicate that the techniques can accurately localize bugs. However, upon further analysis we identify that our techniques are not efficient and/or accurate enough for more complex designs. We therefore discuss and theoretically evaluate new localization techniques that aim to solve the problems of the old techniques.

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. We provide a high level description of our ideas in Chapter 4 and then describe in detail in Chapter 5 how the strategies work. In Chapter 6, we state our experimental methodology and describe the examples we utilized in our experiments. We report our results in Chapter 7 and then discuss them and our techniques in Chapter 8. In Chapter 9, we construct and evaluate new techniques that possess accuracy and/or efficiency improvements over the old techniques. In Chapter 10, we discuss future works for these new 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 accesses to caches, depend on the values of input 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 therefore 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 technique.

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 line evictions or conflicts. The attacker can then utilize these behavioral observations to infer the victim's cache accesses and in turn their secret information. In this section, we provide background on several commonly studied examples of cache attacks.

2.1.1 Flush-based Attacks

Flush+Reload [33] is a flushed-based attack that targets Last Level Caches (LLCs) through the use of shared memory and cache flushing instructions, e.g. `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 location of the victim's data or instructions 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 for a long enough period of time such that the victim can access the cache lines.
3. *Reload* - Reload the monitored cache lines and time the reload latency.

The attacker infers that the victim did not access a cache line during the wait period if they observe a high reload latency. On the other hand, if the reload latency is low, then the attacker knows that victim accessed the monitored cache line. The attacker can then use this knowledge to infer secret information.

Flush+Flush [7] is a similar attack to Flush+Reload. It exploits shared memory and utilizes flush instructions to evict specific cache lines. However, unlike Flush+Reload, Flush+Flush does not directly access/reload memory. Instead, Flush+Flush exploits timing differences in `clflush`'s execution. The attack executes `clflush` in a continuous loop on its targeted cache line. If the flush is quick, then the victim has not recently accessed the cache line. If the flush is slower, then the victim must have 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 [25] 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 [23] 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 victim application's execution.

Prime+Probe 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 [23, 12, 19]. It spies on the victim's cache accesses through the use of eviction sets. Eviction sets are sets of virtual addresses that map to the same cache set. These sets must have enough virtual addresses so that an access to a subset of its addresses will evict an entire cache line. Construction of such sets has been extensively studied and shown to require very little time (linear to quadratic time) [29]. After finding an eviction set, the attacker only needs to:

1. *Prime* - Fill one or more cache sets with its own code or data.

2. *Wait* - Wait for a long enough period of time such that the victim can access the cache lines.

3. *Probe* - The attacker then measures the time to load the same sets of data or code that were previously primed into the cache.

If the memory access latency for one of the lines in the eviction set is high, then it can infer that the victim had accessed that set. Otherwise, the attacker will incur a cache hit and determine that the victim has not accessed that set.

2.1.3 New Attacks

In recent years, non-inclusive and/or sliced caches have become more common, rendering many cache attacks ineffective. These cache designs complicate the mapping between address and cache location, thus impeding the eviction set creation process. A recent paper by Yan et al. [32] proposes an attack that is able to overcome these problems by attacking inclusive cache directories instead. This attack reverse engineers the cache directory structure with special eviction sets and then applies to it common side-channel attacks.

Just this year, Gruss et al. [18] discovered the Collide+Probe and Load+Reload exploits which target AMD processors' caches. These attacks exploit AMD's L1 cache way predictor, which predicts which cache way an address is located in, to efficiently leak sensitive data from the cache.

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 [8] 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. For example, events can be actions such as differing key bits when encrypting the same file. They then use Flush+Reload to construct a profile of each event's memory access patterns. Using this profile and Flush+Reload again, the attacker can match the victim's access patterns with the profile to infer which event occurs.

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 [17] is an unprivileged, practical, simple, and generalizable end-to-end attack that utilizes out-of-order execution and exception handlers to leak sensitive information such as kernel memory. In this exploit, the attacker triggers an exception by accessing privileged

data. The privileged data is then loaded into a pseudo-register for out-of-order instructions to use. These transient instructions access an array, with the index depending on the secret data. During out-of-order execution, the cache state 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 transmit the sensitive index to the attacker.

Spectre attacks [14] 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, resulting in jumps 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 [9] is a set of x86 instructions that allow for hardware-supported 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 and the Random Permutation Cache [20, 30] 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 [26] 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 fuzzing timers to locking sensitive lines in the cache [21, 30].

An effective method for protecting a victim's cache accesses involves partitioning the cache into secure regions controlled by specific processes. The PLCache [30] 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, it cannot be evicted by another process's cache access. Attacks like Flush+Reload, Pericval'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 [13] observes this requirement and thus proposes

to relax the cache inclusion property for critical data used by cryptographic algorithms. By making such data non-inclusive, most cache attacks, which rely on cache inclusivity, 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 [21] achieves this by introducing randomness to the value returned by fine-grained timing instructions (e.g. `RDTSC`).

Other mitigation techniques include the Secure Hierarchy-Aware Cache Replacement Policy (SHARP) and the Prefetch-Obfuscator to Defend Against Cache Timing Channels (PrODACT) [31, 6]. SHARP 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 eliminating inclusion victims, victim data will reside in the L1 cache even after an eviction, in turn confusing the probe/reload steps of cache attacks. PrODACT on the other hand, dynamically analyzes the miss conflict patterns in the LLC and identifies cache sets that are likely being attacked. Once it identifies a possible victim cache set, it uses the prefetcher to obfuscate the victim's 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 subtle timing side-channel that relies on manipulating the cache's LRU policy [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. have shown that an RSA core leaks data through timing variations in its modular exponentiation step [1]. 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 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 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. [22] identify 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)" [22, 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" [22, 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) [28] is a technique that tracks information flow through a CPU's 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, Oberg et al. [22] have shown that GLIFT is able to track all information flows, that is, both timing and functional.

Register Transfer Level Information Flow Tracking (RTLIFT) [2] possesses the same data flow tracking idea as GLIFT, but tracks data at a higher abstraction level: Register Transfer Language (RTL) Level. RTLs model the relationships and signal flows between registers in hardware. RTLIFT's higher abstraction level allows it to achieve 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.

```
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. Example is based on the pseudo code found in Figure 1 of the Clepsydra paper [1].

2.5.3 Multiple Label IFT Techniques

While GLIFT, RTLIFT, and other techniques can track all forms of information flows, both timing and functional, 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). If we wanted to track only timing flows, then we would run into many false positives. For example, the output of an RSA core would always have a high label value since it is always affected by the sensitive key. We would not specifically know if the RSA core has timing variations that depend on the input. Clepsydra and VeriSketch [1, 3] define an IFT approach that solves this problem by separating information flow labels into functional and timing (thus, all data has two labels). The papers define and then prove specific rules that account for the generation, propagation, and blocking of tainted timing flows. Tainted timing flows are generated by unbalanced

conditional assignments to registers that are 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 determine that tainted timing flows propagate to a register if its assigned value or one of its assignment's control signals possesses tainted timing flow. A control signal is one that plays a role in deciding which value should be assigned to a register at clock edges. Tainted timing flows are blocked at a register if at least one of its control signals is non-sensitive and fully controlling. A control signal is non-sensitive if its functional and timing flows are not tainted. 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. This example is based on an example corresponding to Figure 1 in the Clepsydra paper [1, pg. 3]. Note, however, that we extend it 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 a tainted functional flow. 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 is also an automated hardware verification and synthesis technique [3]. That is, this system is able to automatically verify whether an incomplete design written in VeriSketch's sketch syntax conforms to some behavioral and security specifications. The technique does so by using SAT solvers to find an input whose corresponding simulation on the design violates some IFT security properties. If the design fails verification, VeriSketch automatically modifies the sketch in an attempt to satisfy the invariants (synthesis step). If the modified design passes verification, VeriSketch outputs the design in Verilog. However, if the design fails verification, it is modified again in the synthesis step. This cycle continues until VeriSketch outputs or fails to produce a design. Note that it was through the VeriSketch's analysis on the PLCache that we know that the mitigation has a timing-related

vulnerability. After it detected a security bug in the PLCache design, VeriSketch's authors manually found a side-channel in the mitigation that is induced by its eviction policy: when a locked line is accessed by an unlocked line, the locked line becomes the most recently used. This is a problem because while the attacker's data has not been evicted, it is now preferred to be evicted. A modified version of Percival's attack was able to exploit this behavior and induce security breaking timing leakage. While VeriSketch is an effective synthesis tool, it lacks the scalability to large and complex designs. For example, VeriSketch 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, which is not a trivial task. 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 and Caisson [34, 15] implement IFT-based typing systems that are used to verify a hardware design's security. Sapper [16] is an HDL that enforces a design's security by inserting information flow checking logic during code compilation. 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 [2, 1], how to code in different languages than common HDLs, and the more restrictive design patterns enforced by these new languages. Moreover, as these languages require some overhead to enforce IFT rules, their synthesized designs are not as optimized or scalable as

those from common HDLs like Verilog. 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 and efficient localization techniques.

Automatic location and root cause detection has been explored and demonstrated in the post-silicon debugging world. IFRA [24] is an instruction trace-based debugging strategy for CPUs. It utilizes recorders to catch instruction footprints as they leave 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 last few thousand instructions in each recorder. The technique then combines all of the traces offline and then analyzes them against four main heuristics: Data Dependency, Program Control Flow, Load/Store Value, and Instruction Decoding Invariants. In doing this, IFRA is able to determine the location and time of 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 [35] is a pre-silicon verification technique that utilizes backward symbolic execution, a technique in which the engine starts at an error and traverses an execution flow tree backwards, to automatically find and then programatically generate CPU exploits. The exploit 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 backward 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, our work is one of the few, if not only, works in the area of automated timing-related security bug localization. Moreover, it 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 construct two automated simulation-based techniques, one simple (Simple Technique) and one of our own design (Aggregated Graph Walk Technique), that under some assumptions, localize security bugs by analyzing the IFT behavior during faulty simulations.

2. We experiment the two techniques on multiple hardware designs.

3. We discuss both techniques' strengths and weaknesses both with and without some assumptions. We determine that the Simple and Aggregated Graph Walk Techniques are either inefficient or cannot achieve high localization accuracy in more complex designs. Note that in addition to comparing the two techniques between themselves, we also compare them to other naive, "brute force", techniques.

4. Based on the insights gained from the analysis on the two techniques, we propose and theoretically evaluate two new techniques that in theory, can more accurately localize timing-related security bugs. We provide a theoretical discussion on their localization accuracy and efficiency. Finally, we leave the implementation, extension, and empirical evaluation of these two techniques as future works.

CHAPTER 3

DEFINITIONS AND ASSUMPTIONS

We construct two techniques that localize security flaws in Verilog RTL hardware designs by analyzing the IFT signals and behavior 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 conjectures and technique under the following assumptions:

1. We assume that the designer has already instrumented the debugged design with Clepysdra's/VeriSketch's IFT rules [1, 3] 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).

5. No timing flow blockage points, i.e. registers that have fully controlling control signals, exist in the design.

3.2 Definitions

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

- **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 only through at most combinational logic. That is, there is no other register that lies between r_1 and r_2.
- **Register-Wire Adjacency:** A wire w and r are adjacent if one's timing flow label explicitly or implicitly depends on the other's through at most combinational logic. That is, there is no other register that lies between w and r_1.
- **Insecure Path:** A continuous path of adjacent, tainted timing flow carrying, registers that begins somewhere in the design and ends on a register that 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 the 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 without blockage from non-sensitive and fully controlling signals. Thus, when we say bug location, we mean the set of registers where security invariant/monitored label affecting tainted timing flow is 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 our assumptions and 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.

CHAPTER 4

HIGH LEVEL DESCRIPTION

In this chapter, we provide a high level overview of the two timing-related security bug localization techniques. We call the first technique the Simple Technique and the second technique the Aggregated Graph Walking Technique. The Simple strategy consists of two phases: Pre-Processing, Simulation and Candidate Identification. The Aggregated Graph Walking Technique consists of four phases: Pre-Processing, Simulation and Candidate Identification, Graphing, and Filtering. Note that the Pre-Processing and Simulation and Candidate Identification phases are exactly the same in both techniques. We therefore describe the techniques in parallel up to the Simulation and Candidate Identification Phase.

The techniques begin the Pre-Processing phase once a security flaw has been detected and the appropriate inputs are identified. At the start of the Pre-Processing phase, the techniques select from the faulty hardware design registers that affect the monitored labels. The techniques then add logic to the design's RTL code in order to track whether a selected register generates tainted flow during a simulation.

In the next phase, Simulation and Candidate Identification, both strategies simulate the design on faulty inputs. For each input, two traces are kept: one for the registers' taint generation behavior and one for the registers' timing flow labels. Once the techniques finish simulating all inputs, they analyze the traces to determine the set of registers that carry tainted timing flow (call this set R^*) and the set of registers that generate tainted timing flow (call this set R'). Note that R' is exactly the subset of R^* whose registers generate tainted flow. Thus, R' will always contains all the registers that are part of the bug location. Moreover, this set of registers is more specific than the set of registers that carry tainted flow R^*. The Simple Technique therefore stops at this phase and proposes R' as the candidate bug location. The description from now on only applies to the Aggregated Graph Walking technique.

During the Graphing phase, the Aggregated Graph Walking Technique constructs a data

dependency graph that models all timing flow relationships among the registers in the first set. The purpose of this graph is to contextualize the bug location to the designer and serve as a means of reducing false positives in the suggested bug location. The technique constructs the graph by initially starting at the monitored labels and then iteratively walking to adjacent registers that carry tainted flow. This graph, as a result, contains all of the design's insecure paths and ignores some (not all) taint generating registers that are "disjointed". A register is disjointed if the specific taints it carries never touch any monitored labels. Also note that the graph only ignores "some" taint generating disjointed registers because there exists an edge case that cannot be handled by the graph walking algorithm. We further explain disjointed registers and this walking technique's filtering and inaccuracy behaviors towards the end of Chapter 8.

Since the Aggregated Walking Technique ignores some disjointed taint generating registers, it filters out some false positives from the set R^* and thus R'. Therefore, during the Filtering phase, we utilize the graph to filter out false positives in R'. Let D be the set of registers in the dependency graph. The technique then finds and proposes $R' \cap D$ as the potential bug location.

CHAPTER 5

THE TECHNIQUES IN DETAIL

In this chapter, we describe in detail how the Simple and Aggregated Graph Walking Technique work. Like in the high level description, we describe both techniques in parallel up to the Simulation and Candidate Identification phase. The following definitions apply for only this section.

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 $r \in R$ during the simulation on i. It will be of size $|U| \times c_i$.

5.1 Collecting and Comparing Registers

We have restricted our techniques to only collecting traces for register values during simulations. This restriction serves multiple purposes. First, as stated previously, tainted timing

flows can only be generated at registers. Secondly, analyzing and reasoning about registers' behavior is easier as registers propagate their values to each other on at least a one cycle delay. Finally, collecting only the traces of the registers is a significant optimization over collecting both registers and wires. Wires often heavily outnumber registers in a design.

5.2 Pre-Processing

During the Pre-Processing phase, the techniques identify the registers whose timing labels indirectly or directly affect the monitored labels. It places them in R. The techniques then filter 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]. Based on Clepsydra's timing flow logic descriptions and rules, we derive the following logic for timing flow propagation and taint generation for a procedural assignment $r <= r'$:

$$r_{time} <= (r'_{time} \;\&\; !(ns\_full(s_0) \mid ns\_full(s_1) \mid ... \mid ns\_full(s_n))) \mid \tag{5.1}$$

$$(s_{0_{time}} \mid s_{1_{time}} \mid ... \mid s_{n_{time}}) \mid \tag{5.2}$$

$$((s_{0_f} \mid s_{1_f} \mid ... \mid s_{n_f}) \;\&\; !is\_bal(r)) \tag{5.3}$$

In the logic above, the timing flow propagation logic is in lines 5.1 and 5.2, and the generation logic is in line 5.3. 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 r has an unbalanced assignment. Since we want to find the registers that generate tainted timing flow, for each register $u \in U$, the techniques add to the design a register called $u\_is\_generation$, which tracks whether tainted timing flow is ever generated at the register u. The following code is also 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.

5.3 Simulation and Candidate Identification

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

5.3.1 Simulation and Collection

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 label. 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.

5.3.2 Candidate Identification

Consider 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 techniques search the corresponding rows in each T_{i_f} until they find 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 R, the techniques in this step search 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 techniques add 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 Simple Technique proposes R' as the candidate set and also provides the programmer with R^* for the security bug's context. This technique then terminates. However, the Aggregated Graph Walking Technique continues and thus, from now on, we only describe the phases Aggregated Graph Walking Technique.

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

5.4 Graphing

In order to provide the designer with context and better accuracy for the bug location, the Aggregated Graph Walking technique also 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 observe that set R' may contain disjointed registers (as defined at the end of Chapter 4). These registers are false positives because the taint they generate is guaranteed to not affect the monitored labels. Thus, the technique attempts to filter out these false positives by constructing the graph through a walking algorithm that starts at the monitored labels and then walks to the adjacent registers. We now describe the algorithm in detail.

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 therefore conceptually a subset of the registers' 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 add r_d into a global queue.

3. If the queue is not empty, repeat steps 1 to 3 again for each register in the queue that has not already been visited.

4. If the queue is empty or all registers in the queue have been visited, then terminate.

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.

5.5 Filtering Phase

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 filtered out disjointed registers that generate tainted timing flow. Thus, the technique filters the candidate set by selecting only the 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.

CHAPTER 6

EXPERIMENTAL METHODOLOGY

In this study, we tested the strategies' bug localizing 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. In this chapter, we first describe our methodology for validating whether our techniques' suggested sets were accurate. We then detail these examples' configuration and finally discuss our implementation of the techniques for experimentation.

6.1 Validating Candidate Set

Once we had implemented our techniques and were ready to test them on our examples, we needed a procedure that allowed us to check whether the techniques proposed the correct registers. We therefore created a procedure that exhaustively goes through each register in the tested design and checks if it is a part of the the bug location. For each of the examples, we checked the designs' registers for timing generation through an iterative process. We first identified all registers with unbalanced updates U. A subset of U will be the bug location, call it U_s. In order to determine the exact subset, we did the following: let U_s be empty.

```
1  reg_working_dividend_time [8:3]<=(dividend_time [5:0] &
2     ~((~flow_0x7fc928c079c0_t & ~flow_0x7fc928c079c0_time &
3     ~( ( flow_0x7fc928c079c0 ^ flow_0x7fc928c079c0_buf ) ^ (|dividend ^ |reg_working_dividend)))))
4     | {6{flow_0x7fc928c079c0_time}} | {6{flow_0x7fc928c079c0_t}};
5
```

Figure 6.1: Example of timing flow logic in IFT.

```
1  reg_working_dividend_time [8:3]<= {6{flow_0x7fc928c079c0_time}} | dividend_time [5:0];
```

Figure 6.2: The tainted timing flow generation logic has been deleted. Only the propagation logic remains.

For each register $r \in U$,

1. Disabled all registers' tainted timing flow generation logic except for r's. We left the timing flow propagation logic enabled for every register. Figure 6.1 displays a code

example of timing flow logic and Figure 6.2 shows how we disabled its generation logic and left on the propagation logic (labels suffixed with *_time* are the timing labels and those suffixed with *_t* are functional labels). More specifically,

- We disabled its timing flow generation by simply deleting/commenting out the taint generation part of the timing flow code.
- The timing flow propagation logic code is left enabled by simply not deleting it.

2. We compiled and ran the faulty simulations again. If the monitored labels are tainted, then we know that the register is part of the bug location. r is therefore added to U_s.

We finally conducted a check to ensure that the registers U_s were the only bug generating source by disabling their tainted flow generation logic and enabling every other registers' generation logic. All propagation logic was left enabled. If the monitored label was not tainted, then we knew that we identified all of the bug causing registers.

6.2 Multiplier

We experimented on a multiplier that exhibits timing variations when the multiplication of the inputs causes an overflow. Using the validation procedure in Section 6.1, 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 115 random faulty inputs. In all inputs, the multiplicands contained sensitive data and thus had a tainted functional label.

6.3 Divider

The divider design is a subtraction-based division core whose outputs exhibit timing variations depending on the dividend's value. We found with Section 6.1's procedure that 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 inputs carried tainted functional flow. That is, they contained sensitive data.

6.4 AES Cores Injected with Divider

These examples consist of AES cores injected with the divider module described above. 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 and multiple examples with large data loops. 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.

One of our loop examples 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 1". The other loop example has a data loop from an *expand_key_128* module ahead of $a5$, called $a8$, to the buggy *expand_key_128* module $a5$. In this example, $a8$'s (possibly tainted) outputs, when available, are used as $a5$'s inputs. We refer to this example as the "Loop 2" example. In both cases, 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 the Loop 1 case and 87 random faulty inputs for the Loop 2 case.

In all three examples, 6 registers were unbalanced. Moreover, each key and state inputs carried sensitive data.

6.5 RSA Core

This RSA encryption core possesses a timing bug in the modular exponentiation step. Using Section 6.1, 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. $rsacypher.modsqr.modreg1$
14. $rsacypher.modsqr.modreg2$
15. $rsacypher.modsqr.prodreg2$
16. $rsacypher.modmultiply.first$
17. $rsacypher.modmultiply.mpreg$
18. $rsacypher.modmultiply.mcreg$
19. $rsacypher.modmultiply.modreg1$
20. $rsacypher.modmultiply.modreg2$
21. $rsacypher.modmultiply.prodreg2$

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. In this example, we utilized five random inputs to run the technique on.

6.6 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. 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 Section 6.1, are the registers: $cache.MyCtrl.cache\_we$, $cache.MyCtrl.stall\_cycles$, $cache.MyCtrl.wr\_cache\_line\_enabled$, and $cache.Mem32.mem$. 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. In the design, 6 registers were unbalanced.

6.7 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. 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.

3. Used PyVerilog, PyGraphviz, custom Python scripts, and the simulation traces to generate the dependency graph [27, 10].

4. Used a script to find the intersection between the set of registers in the dependency graph and the Simulation and Identification phase's candidate set.

CHAPTER 7

RESULTS

In this chapter, we present the Simple Technique's candidate bug location and accuracy for each example. We also present the Aggregated Graph Walking 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 Aggregated Graph Walking technique's suggested bug locations.

7.1 Multiplier

Table 7.1: 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

7.1.1 Simple Technique

In this example, the faulty simulations executed for a total of 9 cycles. As we can see in Table 7.1, only two registers out of the eight that affect the monitored labels ever generated

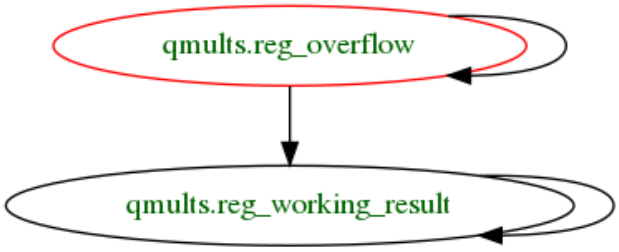

Figure 7.1: The multiplier example's register dependency graph.

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 of the Simple Technique.

7.1.2 Aggregated Graph Walking

The two registers above were also the only ones to carry tainted flow, which means that the dependency graph in Figure 7.1 only has two registers. Thus, during the Filtering phase, the Aggregated Graph Walking 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. Note that the set of unbalanced registers in the design is all the registers found in Table 7.1 except for *qmults.reg_multiplicand_temp*.

7.2 Divider

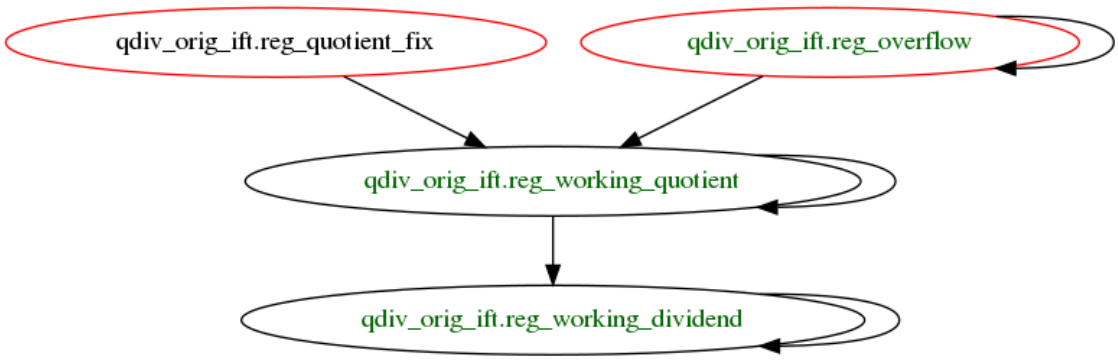

Figure 7.2: The divider example's register dependency graph.

Table 7.2: 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

7.2.1 Simple Technique

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 7.2). Of those four registers, $qdiv\_orig\_ift.reg\_working\_quotient$, $qdiv\_orig\_ift.reg\_working\_dividend$, $qdiv\_orig\_ift.reg\_overflow$ were accurately proposed by the Simple Technique as candidates as they generated tainted timing flow.

7.2.2 Aggregated Graph Walking

The Aggregated Graph Walking technique utilized the four taint carrying registers 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 Filtering 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 $qdiv\_orig\_ift.reg\_working\_dividend$.

Table 7.3: 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

7.3 AES Cores Injected with Divider - Base Example

7.3.1 Simple Technique

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 7.3 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 Simple Technique.

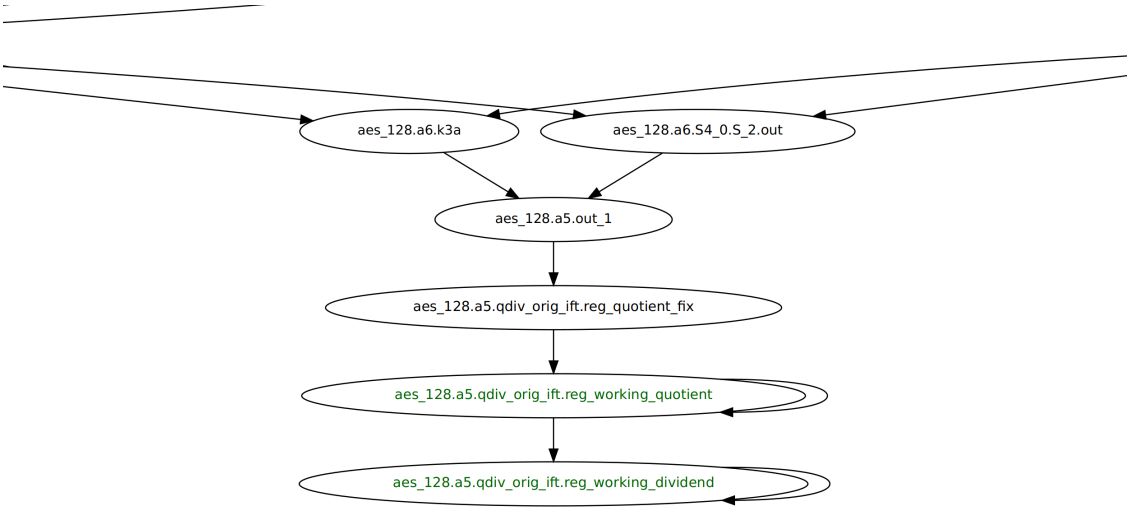

Figure 7.3: The AES with divider base example's register dependency graph (truncated).

7.3.2 Aggregated Graph Walking

Figure 7.3 shows the bug location part of the dependency graph generated from the 134 taint carrying registers 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 Filtering 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 Aggregated Graph Walking Technique accurately determined the bug location of this design.

7.4 AES Cores Injected with Divider - Loop 1 Example

7.4.1 Simple Technique

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 reg-

Table 7.4: The IFT timing behavior of monitored label affecting registers in the AES with divider - Loop 1 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

isters that affect the monitored labels carried tainted flow. Part of Table 7.4 displays the bug location relevant registers and some other registers that carried tainted flow. It shows that earlier registers such as *aes*_128.*a*2.*out*_1 carried tainted flow. Similar to the base example, *aes*_128.*a*5.*qdiv_orig_ift.reg_working_quotient* and *aes*_128.*a*5.*qdiv_orig_ift.reg_working_dividend* generated tainted flow. They were thus the candidate set, which accurately predicted the bug location.

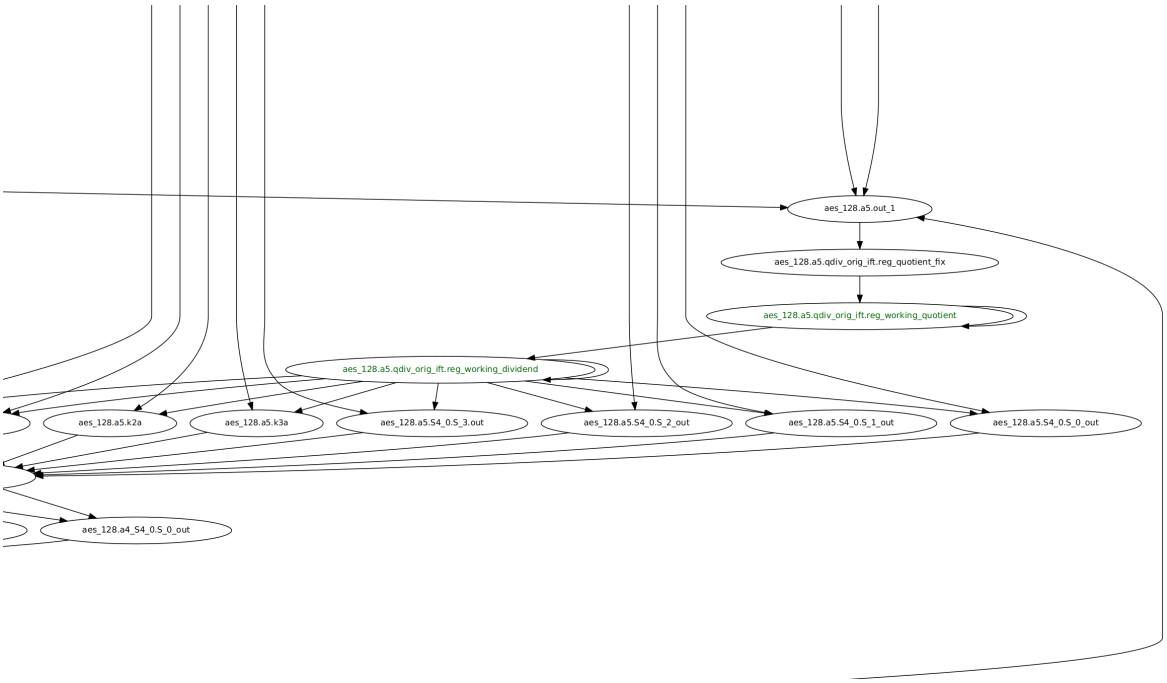

Figure 7.4: The AES with divider - Loop 1 example's register dependency graph (truncated).

7.4.2 Aggregated Graph Walking

The Aggregated Graph Walking 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 the Filtering phase, 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.

Table 7.5: The IFT timing behavior of monitored label affecting registers in the AES with divider - Loop 2 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.a5.k0a	Yes	No
aes_128.a5.k1a	Yes	No
aes_128.a5.k2a	Yes	No
aes_128.a5.k3a	Yes	No
⋮	⋮	⋮
aes_rf_state_out_time	Yes	No
aes_128.a10.out_1	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

7.5 AES Cores Injected with Divider - Loop 2 Example

7.5.1 Simple Technique

The simulations in this example ran for 45 cycles. The tainted timing flow during the faulty simulations propagated out of $a8$ and looped back to the beginning of $a5$, which in turn tainted some labels that affect the buggy register's label. As a result, 175 of the 413 registers tracked carried tainted flow. Table 7.5 displays the information our technique gained from analyzing the register traces. It shows that earlier registers such as $aes\_128.a5.k0a$

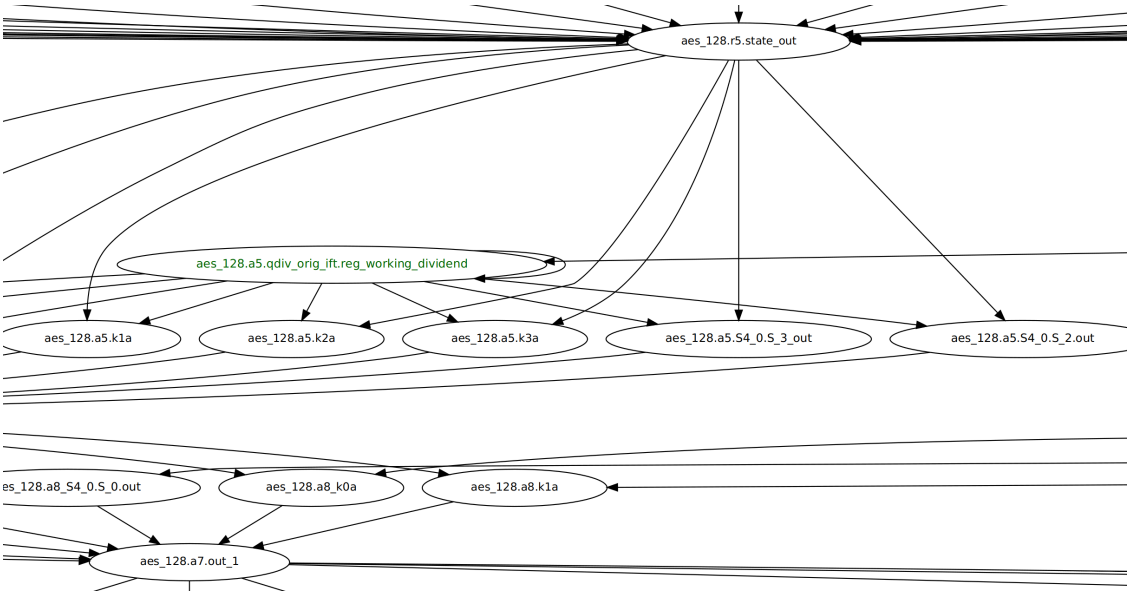

Figure 7.5: The AES with divider - Loop 2 example's register dependency graph part 1.

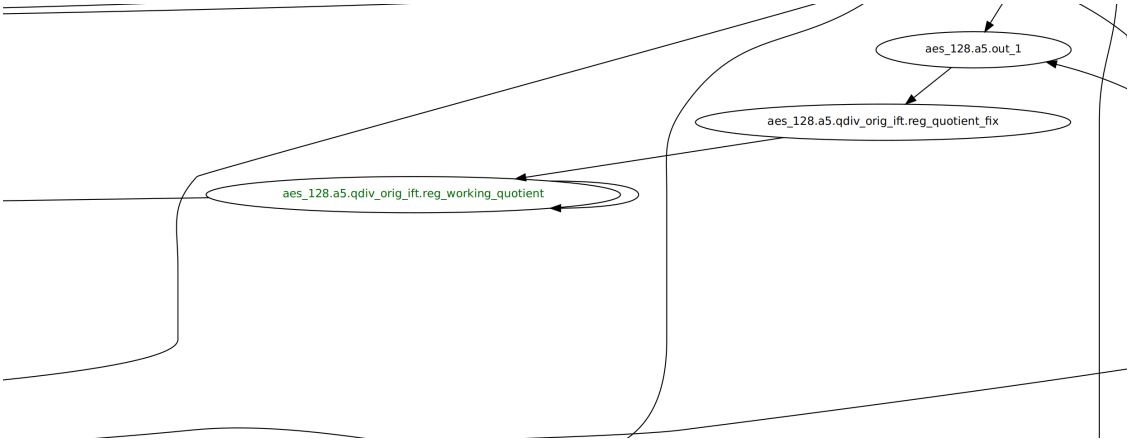

Figure 7.6: The AES with divider - Loop 2 example's register dependency graph part 2. Note that *aes_128.a5.qdiv_orig_ift.reg_working_quotient* depends on *aes_128.a5.qdiv_orig_ift.reg_working_dividend*

carried tainted timing flow. The registers *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 comprised the candidate set, which matched the bug location exactly.

7.5.2 Aggregated Graph Walking

The Aggregated Graph Walking Technique generated the dependency graph displayed in Figures 7.5 and 7.6. The figures contain the bug location part of the dependency graph generated from the 175 taint carrying registers. Note that the two highlighted registers in the figures are directly connected to each other like they are in the previous AES examples. $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_dividend$ again depends on a register because of the data loop. All registers in the candidate set were checked against the graph during the Filtering phase and again, both registers were in the graph. $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_quotient$ and $aes\_128.a5.qdiv\_orig\_ift.reg\_working\_dividend$, were therefore identified as the bug location. Thus, this technique accurately determined the bug location of this design.

7.6 RSA

7.6.1 Simple Technique

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 registers carried and generated tainted flow. The candidate set identified by the Simple Technique was therefore all 21 registers in the design. The Simple Technique therefore accurately found the bug location in this example.

7.6.2 Aggregated Graph Walking

The Aggregated Graph Walking Technique generated a dependency graph that contained all 21 registers as well. Figure 7.7 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. Thus, our technique accurately selected all 21 registers as the bug location.

Table 7.6: 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

Table 7.7: 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

7.7 Cache

7.7.1 Simple Technique

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 Simple Technique found

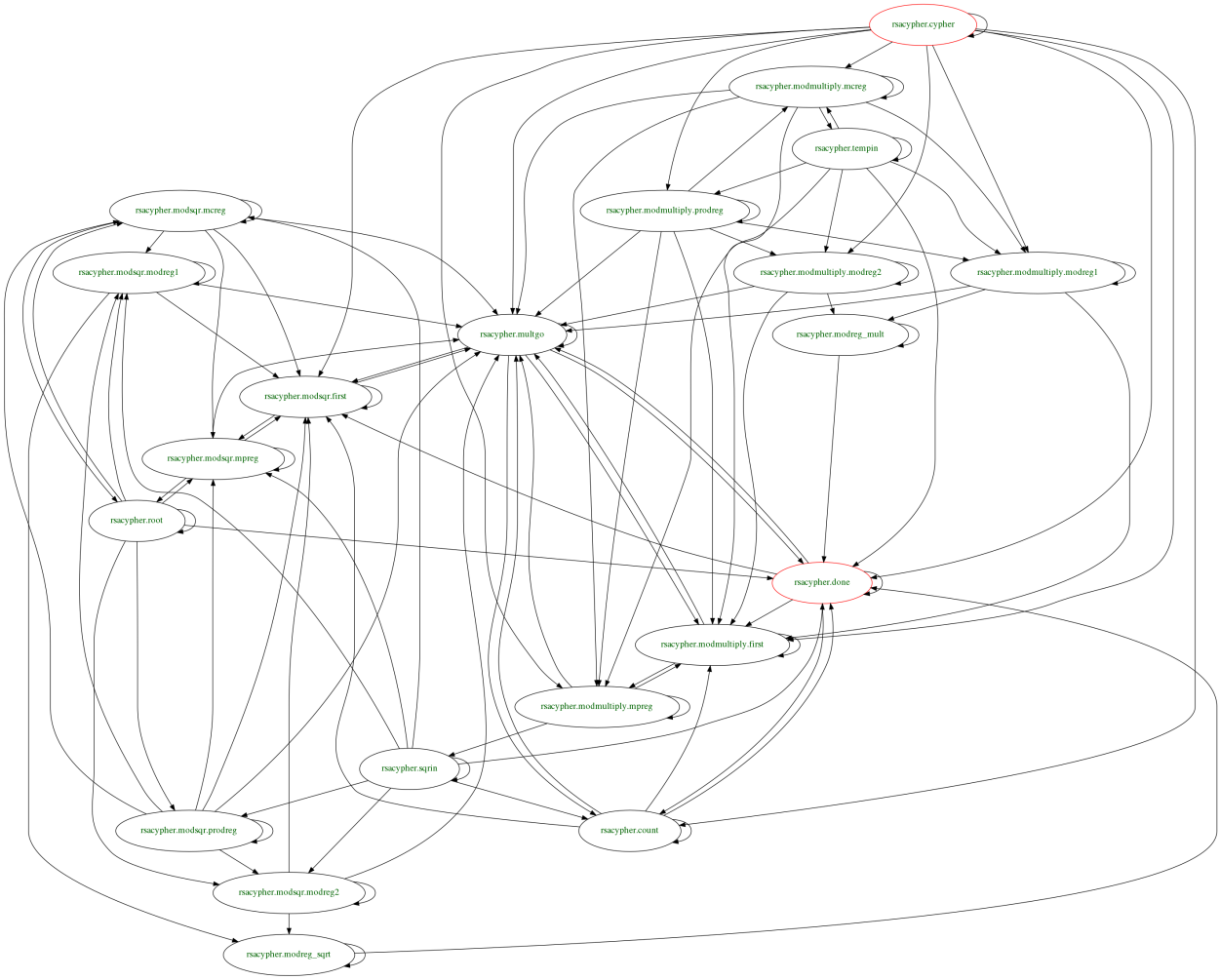

Figure 7.7: RSA example's register dependency graph.

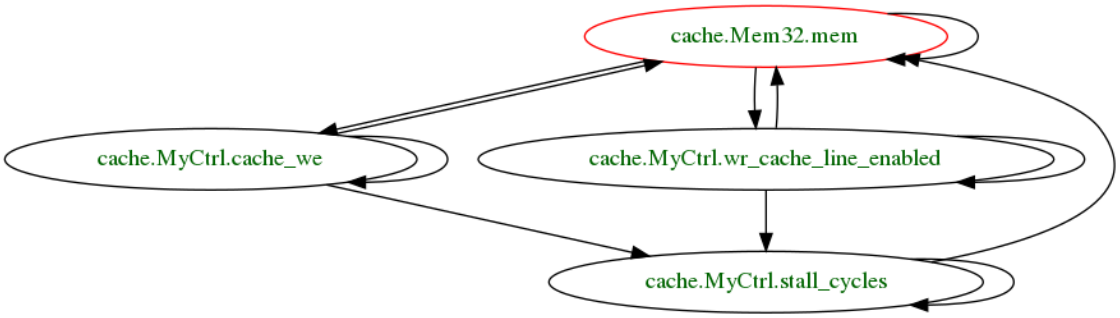

Figure 7.8: Cache example's register dependency graph.

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 7.7). This technique therefore accurately identified those four registers as the bug location.

7.7.2 Aggregated Graph Walking

The dependency graph generated by the Aggregated Graph Walking Technique shown in Figure 7.8 contains the four taint generating registers with a multitude of cycles, indicating overlapping insecure paths. No registers were filtered out from the candidate set as the graph contained all four timing flow generation registers. This technique therefore accurately selected the four registers above as the bug location.

CHAPTER 8

DISCUSSION

In this chapter, we discuss our results from and the various insights we gained about the techniques. We then discuss the practical usage of our techniques and compare them to other localization approaches. We finally describe the major problems with each technique to motivate the need for a better localization technique.

8.1 Results

Our techniques accurately found the location of each example's timing-related security bugs. That is, the techniques' candidate sets of registers were exactly the bug generating locations. Note that the dependency graphs generated by the Aggregated Graph Walking Technique were unneeded to achieve 100% accuracy for our experimental examples. However, in many different examples, this technique achieves greater accuracy than the Simple Technique's. We explain why later in this chapter (Sections 8.6 and 8.7).

8.2 Optimal Number of Faulty Simulations

We utilized varying amounts of faulty simulations during the 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 technique 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.

8.3 Using the Techniques to Fix Bugs

The Simple Technique provides the designer with a candidate set of registers and the set of taint carrying registers (for context). 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.

The Aggregated Graph Walking Technique provides 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 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. We therefore believe that the dependency graph is important because it exposes these "choke points". The designer therefore should use both the candidate set of registers and dependency graph produced by the technique when they wish to fix a bug.

8.4 Technique Automation

While our implementation of these techniques involved some manual effort, the strategies as a whole can be automated with existing tools today. We mostly utilized scripts to compile,

run, and log the simulations. The trace analysis and dependency graphing were all implemented with separate Python scripts. The manual effort mostly involved post processing and formatting the dependency graphs. Thus, the techniques can be automated by simply combing these scripts with some additional software effort to conduct the manual work.

8.5 The Techniques Compared to More Naive Approaches

The most naive localization approach involves the use of checkpoints, wave forms, and other debugging tools to manually track the propagation of a tainted flow. In many designs, the generated taint can flow to every register later in a design, e.g. in the RSA or the AES with divider examples, thus exponentially increasing the number of paths that need to be checked. The programmer is also limited to analyzing one simulation at a time. Therefore, when the programmer debugs large designs, they have to spend great effort manually analyzing a large number of paths to localize the bug. Our techniques are therefore much better than this approach because they can be automated. Moreover, both techniques are more efficient because they effectively analyze many faulty simulations at once.

A second approach could involve automatic backtracking that starts at the monitored labels and then walks backwards along tainted timing label paths. The only advantage that brute force backtracking has over our techniques is that it guarantees 100% accuracy in all cases. Brute force backtracking, however, would not be as scalable as our techniques because, as mentioned earlier in this section, the generated taint can flow to every register later in a complex design. The backtracking algorithm would have to consider a huge number of tainted paths in large and complex designs, which the Simple Technique does not. The Simple Technique's run time mostly depends on the number of registers in a design. Note that the Aggregated Graph Walking Technique conducts a form of backtracking when it generates the dependency graph. However, this backtracking is executed once whereas brute force backtracking must be conducted on each simulation. If there are different insecure paths across simulations, then the backtracking technique would have to backtrack along a

multitude of paths for all simulations. Note that the Simple Technique also only executes once.

Another naive approach would involve the using the procedure from Section 6.1 to exhaustively check every unbalanced register for taint generation. While it can determine the bug location with 100% accuracy, it is very inefficient. All of the simulations must be executed for each unbalanced register. Our techniques, while less accurate, are more efficient as the simulations only need to be run once. Note that by "less accurate" we mean that our techniques may suggest as candidates some locations which are not actual buggy locations. They will never miss any registers buggy locations. Even if the designer, after using our techniques, wanted to use this brute force method to verify the candidate set, they would have fewer registers to check. For example, in the divider, the candidate set only had 3 registers while the number of unbalanced registers was 7. The candidate sets in the cache and AES with divider examples were also smaller than the corresponding unbalanced sets (4/6 and 2/6 respectively).

Finally, a perfectly accurate strategy would involve uniquely identifying tainted flows by their originating registers and then implementing logic to track them as they propagate to other registers in a design. Each register and monitored label s would have some data structure that keeps 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 would propagate to the monitored labels and thus could be quickly and easily identified. While this strategy sounds great on paper, its implementations are very inefficient. One possible implementation makes use of arrays to store the unique tainted flows at each register and monitored label. However, because the timing flow label of a register is potentially affected by a multitude of other labels (e.g. control signals' labels), the arrays must be prohibitively large. Moreover, merging the arrays of two or more signals is extremely slow because new memory must be allocated for each merge and extra computations are needed to deduplicate elements in the merge's output array. Another implementation could involve the use of

pairing functions to encode unique taint IDs into a single value for each register and monitored label. Pairing functions bijectively map a pair of numbers to another number, thus allowing for the inversion of the end result to a unique pair of values. The first problem with pairing functions is that the output values blow up very quickly. The memory footprint would therefore blow up as in the array-based approach. Moreover, there is no good way to avoid duplicates in a pairing function, which compounds the memory overhead problem. The second problem with pairing functions is that they are computationally complex. For example, the Cantor pairing function requires a division, a multiplication, and multiple addition operations. Executing this operation for each register assignment would make simulations on the augmented design infeasible. Our techniques on the other hand, have relatively light memory footprints. The techniques are also not computationally inefficient to the point of being infeasible.

8.6 Problems with the Naive and Aggregated Graph Walking Techniques

8.6.1 Simple Technique

While the Simple Technique was able to localize bugs with 100% accuracy on the example designs, the designs were too small and simple to expose the technique's problems. In differently structured designs, the technique will produce disjointed register false positives. For example, consider the hardware design simulation found in Figure 8.1. 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 8.1, then $r1$ would be a disjointed register that generates tainted flow. The Simple Technique would incorrectly suggest $r1$ as part of the bug location since it generates tainted flow. Consider another example in Figure 8.2. Again, if the design only has faulty simulations like the two in

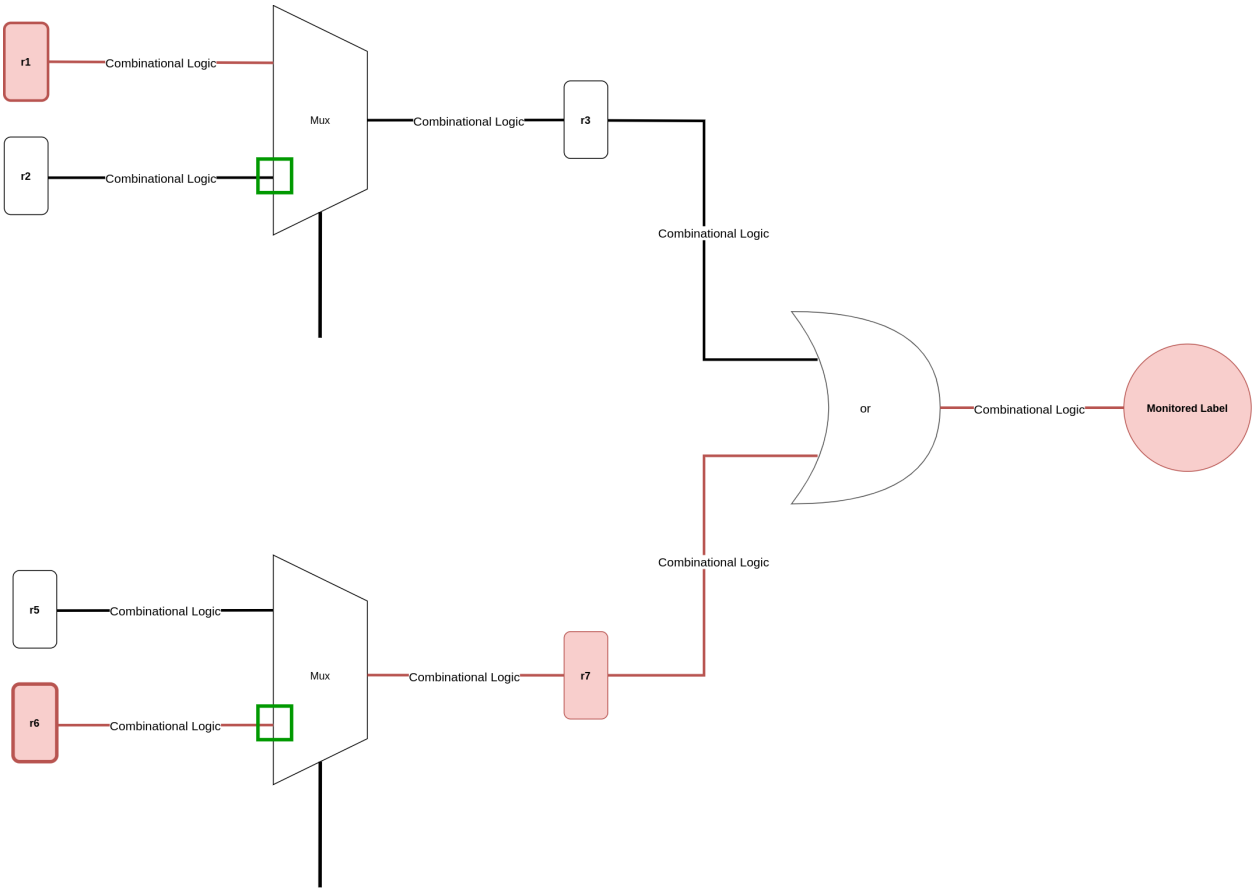

Figure 8.1: A faulty simulation on a theoretical design that results in disjointed register false positives in the Simple Technique's candidate set. Each rounded rectangle and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box.

Figure 8.2, then the control flow behavior at both multiplexers 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 8.1 and 8.2, then disjointed taint generating registers will be produced and the Simple 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.

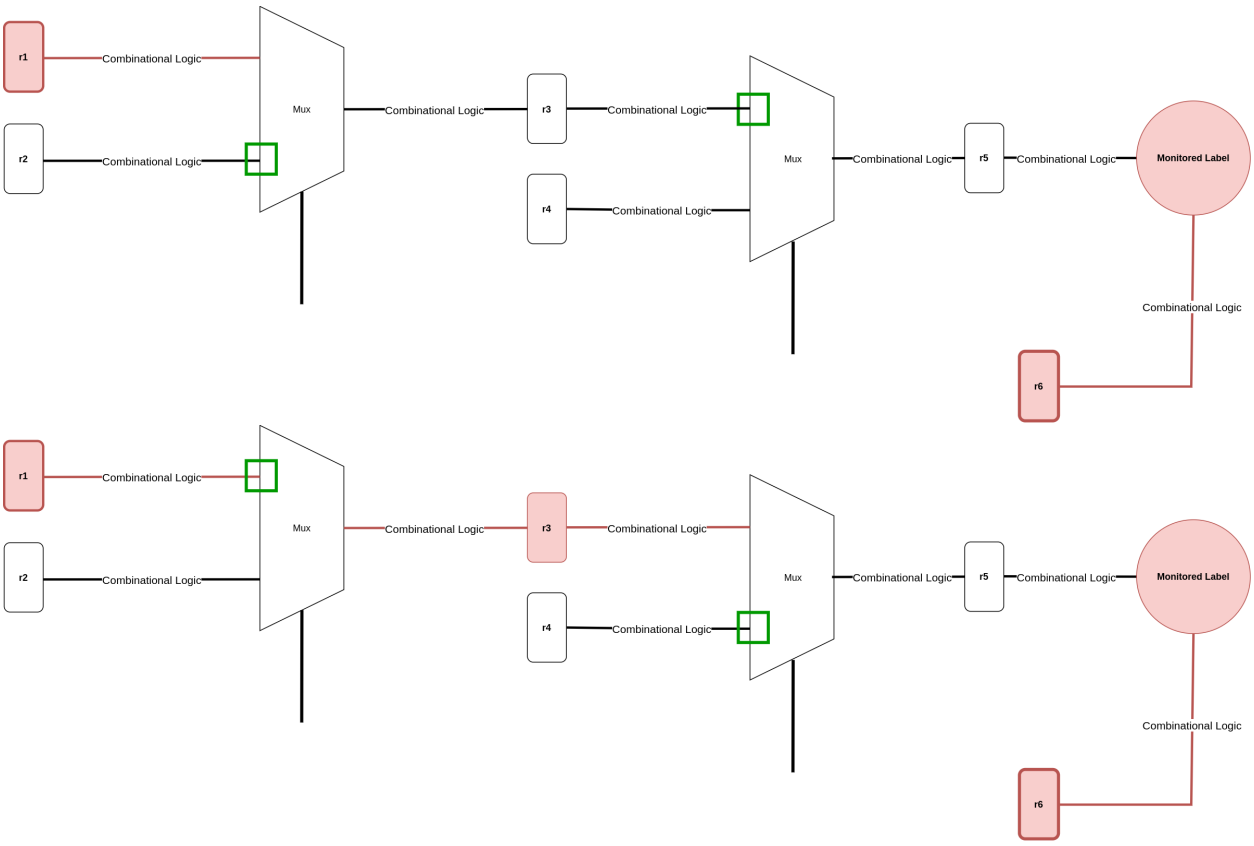

Figure 8.2: Two faulty simulations (a) and (b) on another theoretical design that result in disjointed register false positives in the Simple Technique's candidate set. Each rounded rectangle and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box.

Note that the Aggregated Graph Walking Technique filters out the type of disjointed registers found in Figures 8.1 and 8.2. That is, if there is a non-taint carrying register between a disjointed register 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, as we stated in Chapter 5, in many cases, the graph walking technique will have better accuracy than the Simple Technique. We did not see such an improvement in our tested examples because their simulations produced no disjointed registers.

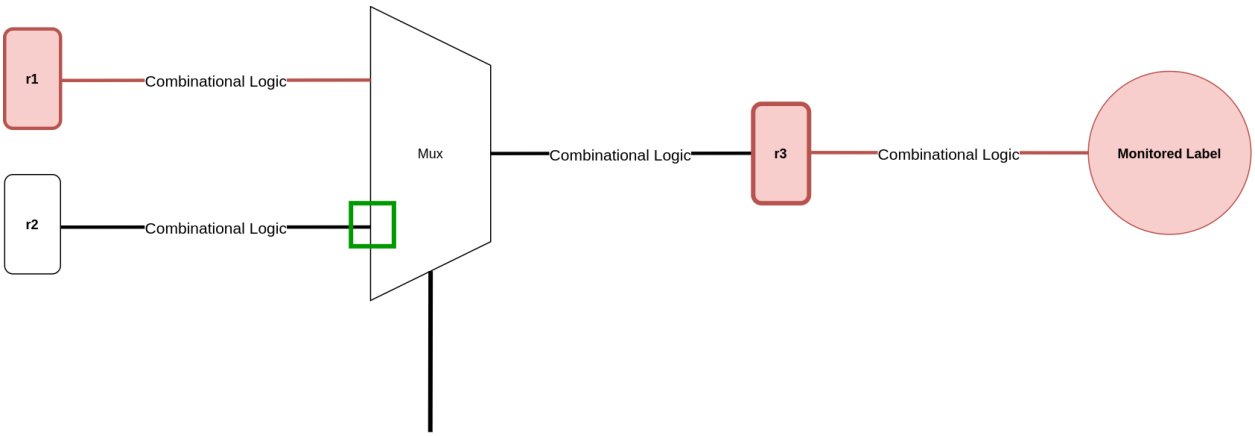

Figure 8.3: A faulty simulation on a theoretical design that results in disjointed register false positives in the Aggregated Graph Walking Technique's candidate set. Each rounded rectangle and each line represents combinational logic. Taint carrying combinational logic and registers are highlighted with red. Taint generating registers are indicated by a thick border. The input chosen at each multiplexer is identified by a green box.

8.6.2 Aggregated Graph Walking Technique

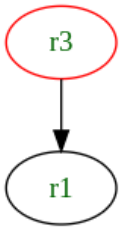

Figure 8.4: Dependency graph produced from the example in Figure 8.3

As discussed in the last paragraph of the previous subsection, the Aggregated Graph Walking Technique can be more accurate than the Simple Technique (e.g. in examples like Figures 8.1 and 8.2). However, it is not invulnerable to disjointed register false positives caused by control flow behavior at multiplexers. If a 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 Aggregrated Graph Walking Technique will add any register along P including the disjointed register to the dependency graph. It will also incorrectly propose d as the bug location. This problem is a consequence of 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 8.3. 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 8.4 and thus incorrectly propose $r1$ as part of the bug location. Thus, better techniques are required to handle disjointed register false positives caused by specific multiplexer control flow behavior.

Finally, while this technique is more efficient than automatic backtracking (as discussed in Section 8.5), it is still relatively inefficient compared to the Simple Technique. As noted above, this technique still conducts a form of backtracking which may take a long time to complete in complex and well interconnected designs.

8.7 Introducing Timing Flow Blockage

Earlier in the paper, we made the assumption that timing flow blockage points do not exist in a design. However, in reality, they exist in a multitude of hardware designs (e.g. in the form of counters). In order to fully discuss timing bug localization, we must therefore lift the no timing blockage assumption. The problem with lifting this assumption is that it leads to a disjointed register problem similar to the one described in the previous section. For example, we tested our technique on 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 $a2$ that the $a5$ 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.

Table 8.1 contains the generation and taint carrying behavior of the registers in this

Table 8.1: "Carries Tainted Timing Flow?" and "Is generation?" behavior from adding a blockage point to the AES with divider base example.

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.a5.qdiv_orig_ift.reg_overflow	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

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 Simple 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 in these two additional registers were always blocked by $aes\_128.a2.qdiv\_fix\_ift.reg\_quotient\_fix$, meaning that they were disjointed registers. On the other hand, the Aggregated Graph Walking Technique generated exactly the same register dependency graph as the one in 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 divder base example. This graph walking technique technique therefore achieved 100%

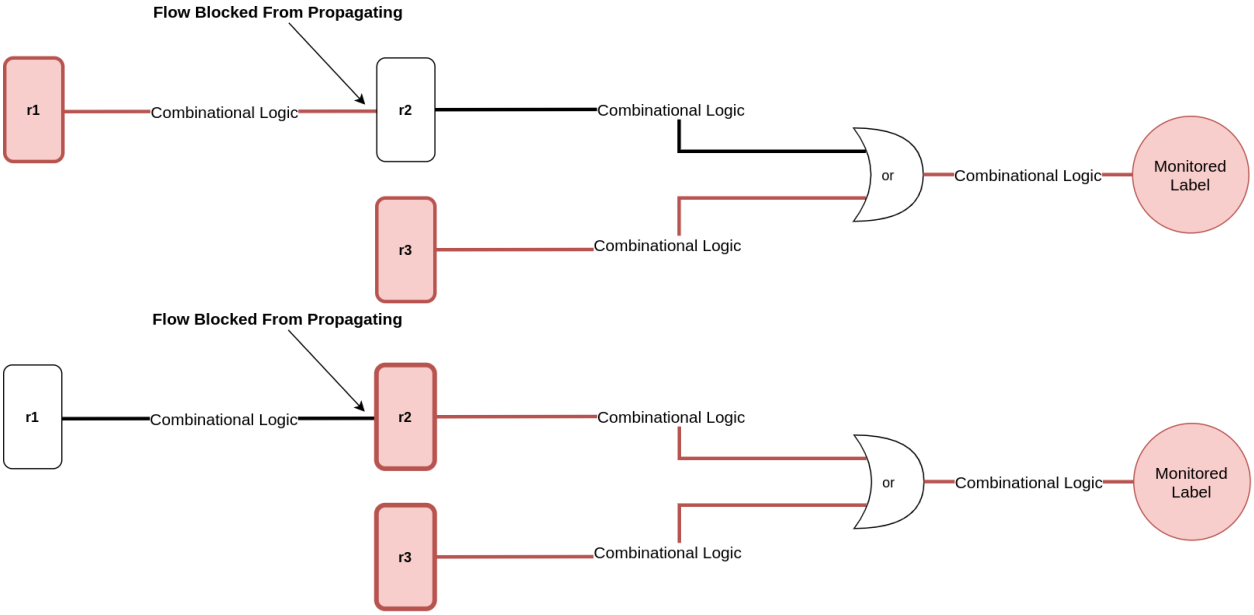

Figure 8.5: Two faulty simulations (a) and (b) on a theoretical design with blockage points that result in disjointed register false positives in the Aggregated Graph Walking Technique's candidate set. Each rounded rectangle and each line represents combinational logic. Taint carrying combinational logic 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.

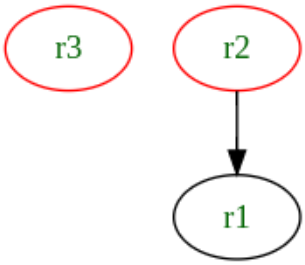

Figure 8.6: Dependency graph produced from the example in Figure 8.5

accuracy in an example with blockage points.

However, like in the previous section, the Aggregated Graph Walking 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 8.4. In the first simulation (Figure 8.5 (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 8.5 (b)), $r1$ no longer generates tainted flow. Tainted flows are generated at $r2$ and $r3$ and then propagate to the monitored labels. The Aggregated Graph Walking algorithm would produce the dependency graph found in Figure 8.6, which mistakenly identifies $r1$ as part of the bug location. Extensive analyses on hardware designs with timing flow blockage are therefore required to fully understand the prevalence, behavior, and consequences of timing flow blockage points. Moreover, better techniques are required to handle timing flow blockage points.

CHAPTER 9

NEW TECHNIQUES

In this chapter, we utilize the insights gained from the Simple and Aggregrated Graph Walking Techniques to present two new techniques that can greatly reduce the false positives generated by the control flow (multiplexers) and timing blockage behaviors we identified in Chapter 8.6 and 8.7. Moreover, we compare these techniques' efficiency with the Aggregated Graph Walking Technique's. We argue that these techniques are as or even more efficient than the graph walking technique. Note that other than the no timing flow blockage assumption, the assumptions we made in Chapter 3 still apply.

9.1 Probabilistic Pathing

We observed in the previous chapter 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.

9.1.1 Detailed Description

This technique first executes a modified Simple Technique. The Simple 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 Simple Technique tracks the values of all timing

flow blockage logic in the design. We consider each 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 Simple Technique, the technique:

1. Determines all paths from r to the monitored labels. 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.

9.2 Probabilistic Pathing Evaluation

In this section, we provide a theoretical evaluation of the Probabilistic Pathing Technique's accuracy and efficiency. For the accuracy evaluation, we describe how the programmer can analyze this technique's results to achieve 100% accuracy on the false positive examples found in Chapter 8.

9.2.1 Accuracy

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 8.1. The probability of the only path from $r1$ to the monitored label, $r1 \rightarrow r3 \rightarrow monitored\_label$ 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 $r6 \rightarrow r7 \rightarrow monitored\_label$ 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 8.2's example is slightly different from Figure 8.1'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, $r1 \rightarrow r3 \rightarrow r5 \rightarrow monitored\_label$ is 1/4 because the chance of $r1$ being chosen at the leftmost multiplexer is 1/2 and the chance of $r3$ being chose 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.

We now consider the example in Figure 8.3. 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 leftmost 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.

The example described in Table 8.1 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.

Finally, the example described in Figure 8.5 is an example with a blockage point that causes false positives in the Aggregated Graph Walking 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.

9.2.2 Efficiency

The computational efficiency of this technique is about the same as the Aggregated Graph Walking Technique. This is because both techniques must traverse the design's dependency/control flow graph. Both techniques must find the paths starting from the candidate registers and ending at the monitored labels. Note that the register scoring computations should be relatively quick (simple arithmetic) and thus not significantly affect the run time of the technique.

The memory footprint of this new technique is slightly higher than the footprint of the graph walking technique because the tracing of blockage logic and controller signal values is required.

9.3 Bloom Filter Approach

This technique revives the idea of tracking the propagation of taints uniquely identified by their originating location throughout a design. By tracking how specific flows propagate to the monitored labels, we can avoid the explicit tracing of multiplexer and blockage point behavior. We first discussed this idea in Chapter 8.5. The naive techniques that utilized this idea were very inefficient. However, by allowing for a small false positive rate, this new technique is much more efficient. This technique utilizes Bloom filters [4] to track the unique taints residing in registers' timing labels and the monitored labels. A Bloom filter is an array of bits that stores elements for later queries. Tied to each filter is a set of k hashing functions that determine how an element is added to the filter. When an element is added to the filter, it is first put through the k hashing functions to determine k bit locations in the filter. Those k locations are set to 1. When that same element is searched for in the filter, the searching entity takes the k hashed locations and determines that the element is in the filter if the locations all have the value 1. 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. Note that there exists an optimal k that minimizes the false positive rate given the number of elements in and the size of the filter. Bloom filters are also not affected by duplicate additions since the hashing of that element will always return the same bit positions. Moreover we can construct a new filter that contains the union of the elements residing in two filters by simply bitwise or'ing them. Finally, Bloom filters will never produce false negatives. That is, if we search for an inserted element, it is always guaranteed to be found.

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.

9.3.1 Detailed Description

The Bloom Filter Approach first executes the Simple 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 m (with upper bound of 256 bits) and number of hash functions k. 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% [11]. Then for each register $r \in R'$, we generate a random ID number and then hash it k times to determine the location of r's bits in the filter. From these k locations, we generate a $bloom\_filter\_id$ of size m for r. Moreover, r is given a Bloom filter label of size m called r_bloom_filter. For each of r's procedural assignments $r <= u$, this technique adds the following logic to the design:

$$r\_bloom\_filter <= ((\text{all controllers are sensitive or not fully controlling}) ? \quad (9.1)$$

$$(u'_0\_bloom\_filter \mid u'_1\_bloom\_filter \mid ... \mid u'_n\_bloom\_filter) : 0) \mid \quad (9.2)$$

$$(s_{r_0}\_bloom\_filter \mid s_{r_1}\_bloom\_filter \mid ... \mid s_{r_n}\_bloom\_filter) \mid \quad (9.3)$$

$$((is\_bal(r) \ \& \ (\text{one of } (r <= u)\text{'s control signals carries tainted functional flow})) ? \quad (9.4)$$

$$(r\_bloom\_filter\_id) : 0) \quad (9.5)$$

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.
- $S_r = \{s_{r_0} \mid s_{r_1} \mid ... \mid s_{r_n}\}$ the set of registers that drive the control signals of $r <= u$.
- Correspondingly, $\{s_{r_0}\_bloom\_filter \mid s_{r_1}\_bloom\_filter \mid ... \mid s_{r_n}\_bloom\_filter\}$ 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 $x <= u$:

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

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

$$(s_{r_0}\_bloom\_filter \mid s_{r_1}\_bloom\_filter \mid ... \mid s_{r_n}\_bloom\_filter) \tag{9.8}$$

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 m-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):

$$l\_bloom\_filter <= d\_0\_bloom\_filter \mid d\_1\_bloom\_filter \mid d_n\_bloom\_filter \tag{9.9}$$

where $\{d\_0\_bloom\_filter,\ d\_1\_bloom\_filter, ..., d_n\_bloom\_filter\}$ 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 Simple 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.

9.4 Bloom Filter Approach Evaluation

In this section, we provide a theoretical evaluation of the Bloom Filter Approach's accuracy and efficiency. We also describe how this technique can achieve 100% accuracy on the false positive examples found in Chapter 8.

9.4.1 Accuracy

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 Simple Technique and thus less accuracy than the Aggregated Graph Walking Technique. That is, all of the taint generating registers, even the false positives, can be found in the monitored label's Bloom filter. However, as long as the number of taint generating registers is less 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 Simple and Aggregated Graph Walking Technique. 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 Aggregated Graph Walking Technique's [11]. 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 counterexamples previously provided in Table 8.1 and Figures 8.1, 8.2, 8.3, and 8.5. 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 8.1. 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 8.2. 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.

We now analyze the example in Figure 8.3. 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.

The example described in Table 8.1 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.

Finally, the example described in Figure 8.5 is an example with a blockage point that causes false positives in the Aggregated Graph Walking 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 examples found in Chapter 8, the Bloom Filter Approach can greatly reduce the incidence of false positive registers.

9.4.2 Efficiency

The Bloom Filter Approach is more computationally efficient than the Aggregated Graph Walking Technique. This new approach does not conduct an expensive walk over all propagation paths. Moreover, the 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 graph walking algorithm.

The Bloom Filter Approach possesses a moderate memory overhead over the Graph Walking Algorithm's memory footprint because an additional ID for every taint generating register and Bloom filter for every register must be added to a design. However, this overhead

is acceptable because it should not blow up like those of the array and pairing function-based techniques.

CHAPTER 10

FUTURE WORKS AND CONCLUSION

10.1 Future Works

10.1.1 Implementation and Empirical Evaluation of New Techniques

So far, we have only described and analyzed the new techniques on a theoretical level. Future works should therefore implement these techniques in practice. Moreover, after implementation, future works should evaluate the techniques' accuracy and efficiency on a multitude of designs.

10.1.2 Complexity and Size of Debugged Designs

The designs we debugged in this paper were relatively simple and small, with the most complex hardware being the cache and RSA designs. As we stated earlier, this study's examples were too simple to reveal common false positive cases. The longest running design also only ran on the order of tens of thousands of cycles. Future works on the new techniques should therefore experiment on a multitude of larger and more complex designs. Moreover, many of the new examples should be able to produce the false positives found in the old techniques. This application of the new techniques to these more complex designs should test for both their localization efficiency and accuracy.

10.1.3 Probabilistic Pathing Technique Efficiency Improvements

We determined in Chapter 9 that the Probabilistic Pathing Technique possesses about the same performance as the Aggregated Graph Walking 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 and Aggregated Graph Walking Techniques; 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.

10.1.4 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 technique will not work with functional flow-related bugs because it is specifically built for tracking unique tainted flow generation. However, the Probabilistic Pathing Technique may be generalizable to functional flows. It can calculate the probability for some taint to propagate on a path to the monitored labels. This technique could thus localize functional flow-related bugs by calculating the probability of a taint-carrying path

that ends at the monitored labels. Future works should therefore look into modifying this strategy to tracking tainted functional flows.

10.2 Conclusion

In this paper, we tested two automated IFT-based techniques (Simple and Aggregated Graph Walking) that, under some assumptions, performed remarkably well in localizing a timing-related security bug. However, upon further analysis, we identified that our examples were too simple to expose common false positive cases in those techniques. We also identified the cause of these false positive cases. Using the insights gained from this analysis, we developed two new techniques (Probabilistic Pathing and Bloom Filter) designed to address these problems. We conducted a theoretical evaluation and so far, these techniques appear to be feasible. Thus, in future works we intend to implement and evaluate them on a multitude of insightful examples. In the future, we also hope to see work in improving our new 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] 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`.

[6] 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`.

[7] 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`.

[8] 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`.

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

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

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

[12] 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`.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, Aug. 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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`.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.