Developing Trustworthy Hardware with Security-Driven Design and Verification

by

Timothy D. Trippel

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2021

Doctoral Committee:

Assistant Professor Matthew Hicks, Co-Chair, Virginia Tech
Professor Kang G. Shin, Co-Chair
Research Professor Peter Honeyman
Assistant Professor Baris Kasikci
Professor Euisik Yoon

Timothy D. Trippel

trippel@umich.edu

ORCID iD: 0000-0002-3448-6868

To my parents, Christine and Terrence.

And to my siblings, Caroline and Christopher.

ACKNOWLEDGEMENTS

As my time as a Michigan graduate is student is coming to an end, I reflect on the people in my life that partook in making this experience both possible and exciting.

First I would like to thank my advisor, Professor Kang G. Shin. While I did not start the Ph.D. program as a student in his lab, Professor Shin took a chance on me. He taught me the value of perseverance, and has kept me motivated throughout my studies.

I would also like to thank my co-advisor, Matthew Hicks, for his support and mentorship throughout my academic career. I first met Matt when he was finishing his position as a post-doc at Michigan. During his first year as research staff at MIT Lincoln Laboratory, Matt convinced me to due a summer internship in his group. Little did I know, that internship would set the path for the remainder of my dissertation work. Matt has always kept my best interests in mind, and has taught me how to become a successful researcher.

Thank you to my remaining committee members—Baris Kasikci, Peter Honeyman, and Eusik Yoon—for their feedback and support. A special thank you to Peter Honeyman for always being available to chat, and guiding me through the ups and downs of graduate school. I will miss his end-of-term celebrations, and seeing him out around Ann Arbor.

Thank you to my high school and undergraduate mentors who sparked my interests in scientific research and engineering. A special thank you to Matt Champion from the University of Notre Dame who gave me an early glimpse of academic research as a high school student. Through various successful science fair projects Matt sparked my creativity in engineering and always held me to a high standard. Additionally, I want to thank Chengkok Koh and David Meyer from Purdue University that gave me opportunities to continue

pursuing scientific research and teaching during my time as undergraduate student. Without them, I would have never pursued graduate school to begin with.

Thank you to Kevin Bush, my supervisor and close mentor at MIT Lincoln Laboratory. I first met Kevin when I came to work in his group at Lincoln with Matt Hicks. The internship was such a success I decided to do two more during my graduate studies. Kevin's support both academically and professionally has directly contributed to the success of many projects in my dissertation. I would also like to thank my industry mentors at Google—Alex Chernyakovsky, Garret Kelly, and Dominic Rizzo—for giving me the opportunity to apply my research ideas to real-world commercial products, and for a great virtual internship experience, during unprecedented pandemic times.

I would like to extend a thank you to my early lab mates, Ofir Weisse and Jeremy Erickson, who helped guide me early in the program, and have provided me countless hours of advice as I transition to the next chapter of my career. Additionally, thank you to all RTCL lab members who overlapped with me during my time at Michigan, especially, Kassem Fawaz, Eugene Kim, Yu-Chih Tung, Kyong Tak Cho, Youngmoon Lee, Hamed Yousefi, Arun Ganesan, Dongyao Chen, Chun-Yu Chen, Taeju Park, Mert Pesé, Duc Bui, Juncheng Gu, Jinkyu Lee, Haichuan Ding, Youssef Tobah, Hsun-Wei Cho, Noah Curran, and Wei-Lun Huang.

Additionally, thanks to all the friends I made at Michigan, especially my house mates and colleagues: Alejandro, Alex, Becky, Ian, Jonathan, Jule, Lauren, Max, Mo, and Richard. You made these years fly by with delicious food and good times.

Thanks to my Grandma (June) and Grandpa (Ed) Garrow, and Grandma (Angie) and Grandpa (Jim) Trippel, for being supportive of me and watching me every summer as I grew up. I know if you were all still alive, you would be very proud of what I have accomplished.

Thanks to my Uncle Bob (UB), Aunt Paula (AP), and my cousins Matthew and Sarah for being my second family and hosting me during my numerous summer internships in the Boston area. Thanks for all your support and encouragement and for making me feel at

home every summer. I always looked forward to spending summers on the Cape, enjoying countless hours of boating, golfing, and eating.

Thanks to my older sister, Caroline, and younger brother Christopher. You are the best siblings I could ever ask for. Thank you for sparking my competitive edge that keeps me motivated and brings out the best in myself. You give me someone to look up to, and always set the bar high.

Most importantly, thank you to my parents, Christine and Terrence who have instilled in me the values of perseverance and education from a young age. You taught me I could do anything and become anyone if I put my mind to it, and always encouraged me to take on tough challenges. Thank you for always ensuring I got the best education, and for signing me up for piano and golf lessons. These activities taught me how to focus and never give up. Thank you for exposing me to computers and electronics at a young age and encouraging me to pursue my interests in science and technology. Lastly, thank you for teaching how to enjoy life through cooking and sharing good food, something my house mates and I have continued throughout graduate school.

Additionally, thank you Lauren for being my best friend, and biggest fan. Your hard work and perseverance through your teaching inspires me everyday. Thanks for keeping me active and encouraging me to try new things through our numerous bike rides, ski trips, hikes, and dining endeavors across Michigan. You make time fly by, and I can't wait for our next adventure.

Lastly, thank you to the several funding agencies that supported the work described in this dissertation, including: 1) the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001, 2) the US National Science Foundation under Grant CNS-1646130, 3) the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 1256260, 4) the US Army Research Office under Grant W911NF-21-1-0057, and 5) the Defense Advanced Research Projects Agency. Any opinions, findings, and conclusions or recommendations expressed in this

dissertation are those of the author and do not necessarily reflect the views of the funding agencies.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

Appendix

LIST OF ABBREVIATIONS

ABSTRACT

Over the past several decades, computing hardware has evolved to become smaller, yet more performant and energy-efficient. Unfortunately, these advancements have come at a cost of increased complexity, both *physically* and *functionally*. Physically, the nanometer-scale transistors used to construct Integrated Circuits (ICs), have become astronomically expensive to fabricate. Functionally, ICs have become increasingly dense and feature-rich to optimize application-specific tasks. To cope with these trends, IC designers outsource both fabrication and portions of Register-Transfer Level (RTL) design. Outsourcing, combined with the increased complexity of modern ICs, presents a security risk: *we must trust our ICs have been designed and fabricated to specification, i.e., they do not contain any hardware Trojans.*

Working in a bottom-up fashion, I initially study the threat of outsourcing fabrication. While prior work demonstrates fabrication-time attacks (modifications) on IC layouts, it is unclear what makes a layout vulnerable to attack. To answer this, in my IC Attack Surface (ICAS) work, I develop a framework that quantifies the security of IC layouts. Using ICAS, I show that modern ICs leave a plethora of both placement and routing resources available for attackers to exploit. Next, to plug these gaps, I construct the first routing-centric defense (T-TER) against fabrication-time Trojans. T-TER wraps security-critical interconnects in IC layouts with tamper-evident guard wires to prevent foundry-side attackers from modifying a design.

After hardening layouts against fabrication-time attacks, outsourced designs become the most critical threat. To address this, I develop a dynamic verification technique (Bomber-

man) to vet untrusted third-party RTL hardware for Ticking Timebomb Trojans (TTTs). By targeting a specific type of Trojan behavior, Bomberman does not suffer from false negatives (missed TTTs), and therefore systematically reduces the overall design-time attack surface. Lastly, to generalize the Bomberman approach to automatically discover other behaviorally-defined classes of malicious logic, I adapt coverage-guided software fuzzers to the RTL verification domain. Leveraging software fuzzers for RTL verification enables IC design engineers to optimize test coverage of third-party designs without intimate implementation knowledge. Overall, this dissertation aims to make security a first-class design objective, alongside power, performance, and area, throughout the hardware development process.

CHAPTER I

Introduction

1.1 Motivation

Since the inception of the Integrated Circuit (IC), modern computing capabilities have been closely tied to advancements in hardware. Through the era of Moore's Law [113] and Dennard scaling [44], computer architects realized most performance gains from periodic, exponentially shrinking, transistor sizes. However, as transistors have gotten smaller, they have become increasingly expensive to manufacture. For example, TSMC's latest IC fabrication facility, slated for completion in 2023, is expected to cost $19.6 billion [46].

Similarly, when transistor scaling became challenging due to fundamental physical limits, computer architects responded by increasing the amount of on-chip parallelism, first, through (homogeneous) multi-core designs, and second, through (heterogeneous) domain-specific accelerators [37, 55, 78, 106, 118, 141]. Unfortunately, with increased heterogeneous parallelism, comes increased verification complexity. Today, to completely design and verify an IC of moderate complexity on an advanced silicon node requires an estimated 500 engineering years, or one year's time from 500 engineers working together [53, 92]. Of this time, it is estimated that up to 70% is spent verifying design correctness [47], rather than implementing the design itself.

1.2 Hardware Development Trends

In response to these challenges, hardware development trends have shifted. First, to cope with rising fabrication costs, most semiconductor companies have become *fabless*. While they still have the financial resources to *design* custom hardware, they *outsource* its fabrication. Today, only two companies remain that are capable of fabricating leading edge silicon at scale (≈40,000/wafers a month): Samsung and TSMC[1] [95]. Second, to continue building increasingly complex designs, while maintaining expected time-to-market, semiconductor companies have outsourced portions of the design/verification process by purchasing third-party IP blocks [171].

1.3 Research Challenges & Thesis Statement

Unfortunately, outsourcing combined with the size and complexity of modern ICs presents a security risk. How do we know untrusted third parties will design and fabricate ICs according to specification? In other words, how can we be certain our designs will be free of malicious modifications, i.e., hardware Trojans?

Thesis Statement:

In this dissertation, I present techniques to harden IC designs against the threats of increased outsourcing through security-driven design and verification.

Specifically, I address the security risks associated with outsourcing: 1) *fabrication* and 2) *design*.

[1] As of writing this dissertation, Intel does not fabricate third-party designs at scale, but they have recently announced plans they will in the near future [158].

1.4 Dissertation Contributions

1.4.1 Security-Driven (Layout) Design

In the first half of my dissertation, I tackle the challenge of designing secure hardware in the face of an untrusted foundry. To fundamentally understand how to address this issue, in Chapter III I study the susceptibility of IC layouts to fabrication-time modification, i.e., *fabrication-time attacks*. First, I enumerate the challenges a foundry-side adversary faces when attempting to insert a hardware Trojan into an IC layout. From there, I design, implement, and open-source a framework, called IC Attack Surface (ICAS) [169], that estimates the difficulty—i.e., resources—required to mount a fabrication-time attack. ICAS provides back-end IC design engineers with an automated toolchain to assess the risk of a fabrication-time attack, *before* sending their designs to the foundry. I demonstrate the utility of ICAS by analyzing over 60 real-world IC layouts with and with-out existing layout-level defenses. ICAS results indicate that even with defenses deployed, IC layouts still leave tens to thousands of possible Trojan attack points available for foundry-side attackers to exploit.

To fill these defensive gaps, I proceed in designing and implementing the first routing-centric layout-level defense against fabrication-time attacks—Targeted Tamper-Evident Routing (T-TER)—in Chapter IV. T-TER deploys tamper-evident guard wires—similar to those used for cross-talk reduction [60, 61]—around all sides of security-critical nets in a layout to prevent foundry-side attackers from connecting rogue Trojan wires to, or nearby, them [170]. The analog characteristics of T-TER guard wires are then analyzed post-fabrication, using continuity checks and Time-Domain Reflectometry (TDR), to nondestructively determine if any guard wires were manipulated at the foundry. Using the ICAS framework (Chapter III), I show that with T-TER, back-end design engineers can completely close the fabrication-time attack surface with regard to the security-critical features they care about.

1.4.2 Security-Driven (RTL) Verification

After securing the layout, the next critical threat in modern ICs is the integration of untrusted third-party IP. In the second half of my dissertation, I tackle this threat by proposing a new technique—Bomberman—for vetting third-party IP for hardware Trojans (Chapter V). Unlike prior Trojan verification techniques [58, 184, 202], Bomberman is TTT-specific, and aims to systematically constrict the overall design-time attack surface by provably eliminating the threat of TTTs. To achieve this, Bomberman starts by assuming *all* components in the design are *suspicious* (part of a TTT), and only marks components as benign if they violate any TTT-specific behavioral invariants during verification simulations. As a result, false negatives are *impossible*. Moreover, by carefully crafting verification test vectors, I demonstrate Bomberman's false positive rate is less than 1.2% across four real-world hardware designs, including: an AES accelerator [136], a Universal Asynchronous Receiver-Transmitter (UART) core [121], an OR1200 processor [121], and a RISC-V processor [193].

While Bomberman demonstrates the effectiveness of Trojan-specific verification, its success (false positive rate) is largely determined by a verification engineer's ability to craft test vectors that exercise potential TTT logic. Unfortunately, doing so requires verification engineers have intimate knowledge of the DUT's implementation, which is rarely the case when analyzing third-party designs. To address this issue, and generalize attack-specific verification techniques to automatically discover other behaviorally-defined security vulnerabilities, I propose a new DV technique called *hardware fuzzing* in Chapter VI. In this work, I borrow recent advancements in the software testing community, namely *coverage-guided greybox fuzzing* [20, 201], to optimize RTL DV test generation for maximal design coverage. Rather than re-implementing software fuzzing techniques in the hardware domain, I develop a mechanism to fuzz software models of RTL hardware directly in a design agnostic manner. Moreover, I demonstrate over two orders-of-magnitude improvement in coverage convergence over existing CRV techniques.

1.5 Road Map

The remainder of this dissertation is structured as follows. In Chapter II, I discuss pertinent background information regarding the IC design process, the construction of hardware Trojans, and two hardware supply chain threat models that are the focus of this dissertation. Next, in Chapters III–IV, I present two works that address the threat of outsourcing IC fabrication by making security a first-class design objective during IC layout. Following this, in Chapters V–VI, I present two additional works that address security concerns stemming from rising design complexities by making security a primary objective during RTL verification. Lastly, I conclude with a summary of my dissertation work in the context of my thoughts on future directions in Chapter VII.

CHAPTER II

Background

2.1 IC Design Process

In order to design complex ICs, like the Apple A13 Bionic chip that contains 8.5 billion transistors [43], the design process is broken down into several phases (Fig. 2.1) and heavily augmented with Computer Aided Design (CAD) tools. To design complex ICs, while minimizing time-to-market, hardware designers often first purchase existing IP blocks from third parties to integrate into their designs. Next, designers integrate all third-party IP blocks, and describe the behavior of any custom circuitry at the RTL, using HDLs like Verilog. Next, CAD tools synthesize the HDL into a gate-level netlist (also described using HDL) targeting a specific process technology, a process analogous to software compilation. After synthesis, designers *place* the circuit components (i.e., logic gates) on a 3-dimensional grid and *route* wires between them to connect the entire circuit. CAD tools encode the physical layout in the Graphics Database System II (GDSII) format, which is then sent to the fabrication facility. Finally, the foundry fabricates the IC, and returns it to the designers who then test and package it for mounting onto a printed circuit board.

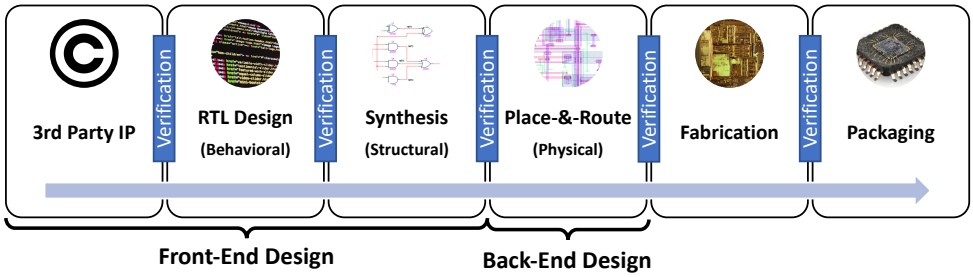

Figure 2.1: **IC Design Process.** The typical IC design process starts with a behavioral description of the design, and ends with a fabricated and packaged chip. All stages in the design process are often heavily augmented with CAD tools.

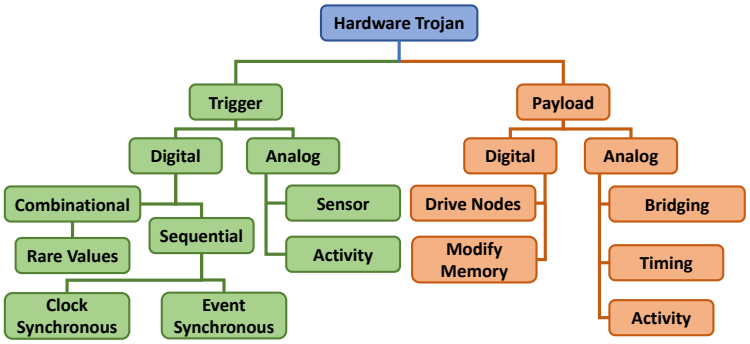

Figure 2.2: **Hardware Trojan Taxonomy.** A hardware Trojan is an undesired alteration to an IC design for the purpose of modifying its functionality. Hardware Trojans can be classified based on the construction of their two main components: trigger and payload [34, 75, 194].

2.2 Hardware Trojans

A hardware Trojan is a malicious modification to a circuit designed to alter its operative functionality [16]. It consists of two main building blocks: a **trigger** and **payload** [34, 75, 169, 194]. Prior work provides hardware Trojan taxonomies based on the type of trigger and payload designs they employ [34, 75, 194]. I adopt this taxonomy, depicted in Fig. 2.2.

2.2.1 Trojan Trigger

The trigger is circuitry that initiates the delivery of the payload when it encounters a specific state. The goal of the trigger is to control payload deployment such that it is hidden from test cases (stealthy), but readily deployable by the attacker (controllable). Triggers are

created by adding, removing, and/or manipulating existing circuit components [90, 142, 162, 196], and can be digital or analog [85, 135, 196]. The ideal trigger—e.g., A2 [196]—achieves stealth and controllability while being small (i.e., requiring few additional circuit components).

2.2.2 Trojan Payload

The payload is circuitry that, upon being signaled by the trigger, alters the functionality of the victim (host) circuit. Like the trigger, the payload can be analog or digital, and has a variety of possible malicious effects. Prior work demonstrates Trojan payloads that leak information [101], alter the state of the IC [196], and render the IC inoperable [142]. *One attribute all documented controllable hardware Trojans have in common is that they must route a rogue wire to, or directly adjacent to, a security-critical wire within the victim IC [169].*

2.3 Threat Models

There are two main threats to the IC supply chain given current economic trends [19, 58, 94, 169, 170, 184, 202]: 1) *out-sourced fabrication* and 2) *out-sourced design*. As a result, in this thesis I focus on two threat models, or IC supply chain attack points: 1) **fabrication-time attacks** and 2) **design-time attacks**, as shown in Figure 2.3.

2.3.1 Fabrication-Time Attacks

In the *fabrication-time attack* threat model, all phases of the IC design process are trusted *except* fabrication (Fig. 2.3A). This threat model stems from the extreme ramp-up costs associated with fabricating leading-edge silicon [94, 95] that make outsourcing IC fabrication a necessity—even for nation states. In line with previous untrusted foundry threat models [101, 135, 162, 169, 196], I assume the worst case: that any fabrication-time modifications are carried out by a malicious actor within the foundry (or any foundry

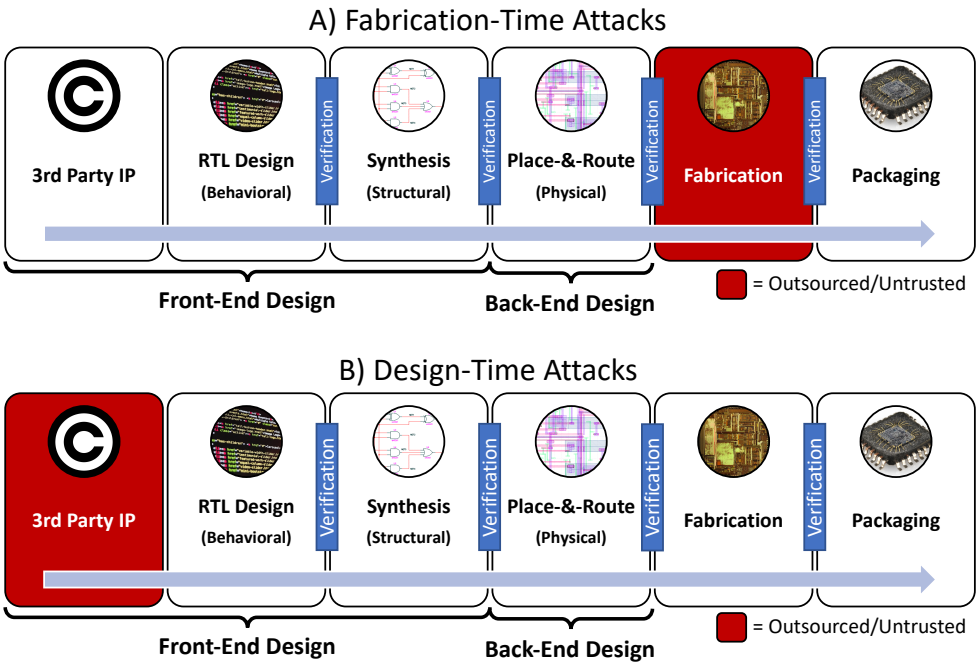

Figure 2.3: **IC Supply Chain Attack Vectors.** Given current economic trends [19, 58, 94, 169, 170, 184, 202], there are two main attack points in the IC supply chain resulting from out-sourcing *fabrication* and *design* respectively: A) *fabrication-time* and B) *design-time*.

partners) that has access to the entire physical layout of the IC in the form of a GDSII file, the output of the Place-&-Route (PaR) phase in Figure 2.1.

Inserting a hardware Trojan at fabrication-time is different from inserting a Trojan during the front-end design. Unlike behavioral or structural-level attackers that maliciously modify the HDL or gate-level netlist, respectively [5, 74, 184], the fabrication-time attacker only has access to the *physical-level* representation of the IC design (i.e., output of the *PaR* phase in Fig. 2.1). Specifically, they must edit the geometric representation of the *circuit layout*, e.g., the GDSII file. While this is more challenging than editing the design at the behavioral- (HDL) or structural-level (netlist), where design specific semantics are more readily interpretable, it is even more difficult to defend. The post-fabrication defender receives a literal black box from the foundry. Comprehensively inspecting each fabricated die to verify the absence of malicious perturbations is infeasible for the most advanced hardware Trojans [196].

2.3.2 Design-Time Attacks

In the *design-time attack* threat model, the untrusted phase is the front-end-design phase, where third party IP is integrated. In order to decrease time-to-market in and maintain feature rich designs, companies favor a reliance on untrusted third parties and large design teams [19]. Moreover, without a trusted front-end design, any result of back-end design and fabrication cannot be trusted. Similar to prior design-time attack studies [69, 99, 151, 181–183, 191], I focus on malicious modifications that are embedded in third party HDL (Fig. 2.3B). I assume that a design-time adversary has the ability to add, remove, and modify the HDL of the core design or IP block in order to implement hardware Trojans. This can be done either by a single rogue employee at a hardware design company, or by entirely rogue design teams. Lastly, I assume that any malicious circuit behavior induced by Trojan trigger *activation* is caught via verification testing [58, 69, 183, 184, 191, 202].

Compared with fabrication-time attacks, design-time attacks are easier to implement, as the attacker has access to semantically rich HDL. However, design-time attacks are also easier to detect. A defender has full visibility into the design and its functionality, and there is no notion of analog behavior. A defender can use heuristics based tools [58, 99, 151, 171, 184, 202] to vet their designs, prior to passing them off to the back-end design phase.

CHAPTER III

ICAS

3.1 Introduction

The relationship between complexity and security seen in software also holds for Integrated Circuits (ICs). Since the inception of the IC, transistor sizes have continued to shrink. For example, compare the $10\,\mu m$ feature size of the original Intel 4004 processor [71] to the $10\,nm$ feature size of Intel's recently announced Ice Lake processor family [4]. Smaller transistors enable IC designers to create increasingly complex circuits with higher performance and lower power-usage. However, continuing this trend pushes the laws of physics and comes at a substantial cost: building a 3 nm fabrication facility is estimated to cost $15–20B [94].

Such costs are prohibitive for not only most semiconductor companies, but also nation states. Thus, **most hardware design houses are fabless**, i.e., while they are able to fully design and lay out an IC, they must outsource its fabrication. Outsourcing combined with the black-box nature of testing a fabricated IC requires fabless semiconductor companies to trust that their physical designs will not be altered maliciously by the foundry, also known as a *fabrication-time attack*. Previous work demonstrates several ways a fabrication-time attacker can insert a hardware Trojan into an otherwise trusted IC [17, 90, 196]. A2 [196] demonstrates the most stealthy and controllable IC fabrication-time attack to date, whereby a hardware Trojan with a complex, yet stealthy, analog trigger circuit is inserted into the

finalized layout of a processor. Even though the inserted Trojan is small, the attacker can trigger it and escalate to a persistent software-level attack (i.e., a hardware foothold [85]) using only user-mode code.

Early work focuses on post-fabrication *detection* of hardware Trojans in ICs [162]. Broadly, there are two classes of detection: 1) side-channel analysis and 2) Trojan-activation via functional testing. Side-channel (power, timing, etc.) analysis [3, 75, 117, 131] assumes that the Trojan's trigger is complex (i.e., many logic gates), and thus noticeably changes the physical characteristics of the chip. For example, inserting the large amount of extra logic required by a complex trigger into a design alters the power signature of the device. Alternatively, Trojan-activation via functional testing assumes that the Trojan's trigger is simple (i.e., few logic gates [17, 90]), and is thus easily activated by test vectors. Unfortunately, layering detection classes is *not* sufficient as it is shown possible to create an attack that is both small and stealthy [196].

To address the gaps left by post-fabrication Trojan *detection* schemes, recent work focuses on pre-fabrication, IC layout-level, Trojan *prevention* [9, 40, 195]. IC layout-level defenses work by:

1. increasing placement & routing resource utilization

2. increasing congestion around security-critical design components.

The lack of resources deprives the attacker of the required transistors needed to implement their Trojan trigger/attack circuits, and the increased congestion around security-critical wires acts as a barrier for the attacker attempting to integrate their Trojan into the victim design. Ideally, defenders utilize just enough resources and create enough congestion such that the attacker cannot implement and insert their attack, while keeping the design routable. Short of that, the added barriers require the attacker to expend significantly more resources (e.g., time) to insert their attack into an IC layout.[1]

[1]Time is the most critical resource for the attacker as IC fabrication is usually bounded in terms of turnaround time.

Two IC layout-level defensive approaches exist: undirected and directed. **Undirected** approaches aim to (probabilistically) increase resource utilization and congestion across the *entire* layout by altering existing place-and-route parameters (e.g., core density [195]) that will likely result in increased resource utilization and congestion. More recently, a line of **directed** approaches have emerged [9, 10] that *systematically* increase utilization of *specific-regions* of the device layer, i.e., nearby security-critical components. Given that it is infeasible to occupy the entire device layer in a tamper-evident manner [9, 10], both classes of approaches may leave IC layouts vulnerable to attack by an untrusted foundry.

To identify gaps in existing defenses and guide future IC layout-level defenses, I design and implement an extensible measurement framework that estimates the susceptibility of an IC layout to foundry-level additive Trojan attacks. Our framework, *IC Attack Surface* (ICAS), estimates resilience in three dimensions that capture the essence and difficulty of inserting a hardware Trojan at an untrusted foundry:

1. **Trojan logic placement:** finding unused space to place additional circuit components

2. **Victim/Trojan integration:** attaching hardware Trojan payload to security-critical logic

3. **Intra-Trojan routing:** connecting the trigger and payload portions of the hardware Trojan

A successful attack requires all three steps.

Using ICAS, I analyze over 60 different IC layouts across three fully-functional ASIC designs: an AES accelerator, a DSP accelerator, and an OR1200 processor. For each layout, ICAS reports the coverage against four additive Trojan attacks [58, 85, 136, 196] that span the digital and analog domain as well a range of attack outcomes. ICAS's analysis reveals that all existing IC layout-level defenses are incomplete, leaving 1000's of opportunities for an attacker at an untrusted foundry to insert a hardware Trojan. An additional finding is

that even though most existing countermeasures do increase the complexity of inserting a hardware Trojan, some countermeasures are ineffective. Lastly, ICAS's analysis suggests that focusing on exhausting resources on the device layer (i.e., transistors) is an incomplete defense; future defenses should also aim to increase congestion around security-critical wires.

This chapter makes the following contributions:

- I propose an extensible methodology that estimates the difficulty of inserting additive hardware Trojans into an existing IC layout by an untrusted foundry.
- I design, implement, and open-source [110, 111] our extensible framework, ICAS, that computes various layout-specific security metrics. The ICAS framework provides an interface to programmatically query the physical layout of an IC (encoded in the GDSII format) to compute various security metrics with respect to attacks-of-interest.
- I use ICAS to estimate the effectiveness and expose the gaps of previously-proposed untrusted foundry defenses by analyzing over 60 IC layouts of three real-world hardware cores.
- I identify future directions for defenses that work in a layered fashion with existing defenses.

3.2 Background

3.2.1 IC Layouts

IC layouts consist of multiple layers. The bottom layers are *device layers*, while the top layers are *metal layers*. Device layers are used for constructing circuit components (e.g., transistors), and the metal layers are used for routing (e.g., vias and wiring). The first stage of PaR is creating a floorplan. Figure 3.1 illustrates an IC floorplan. To create

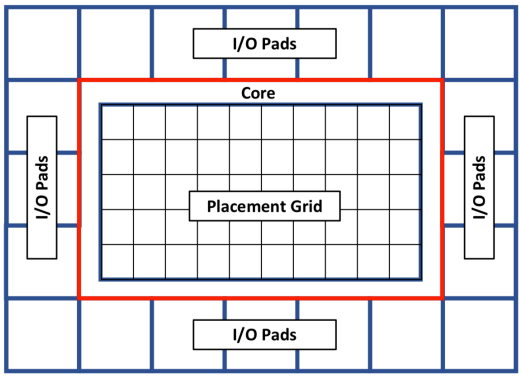

Figure 3.1: **IC Floorplan.** Typical IC floorplan created during the place-and-route design phase. The floorplan consists of an I/O pad ring surrounding the chip core. Within the core is the placement grid. Circuit components are placed and routed within the placement grid.

a floorplan, the dimensions of the overall chip are specified and the core area is defined. Typically a ring of I/O pads is then placed around the chip core, while a placement grid is drawn over the core. Each tile in the placement grid is known as a *placement site*. Circuit components (e.g., standard cells) are then placed on the placement grid, occupying one or more placement sites, depending on the size of the component. Lastly, all components are routed together, using one or more routing layers. The output from the back-end design is a Graphics Database System II (GDSII) file that is a geometric description of the placed-and-routed circuit layout. The GDSII file is then sent to a fabrication facility where it is manufactured. The final step is testing and packaging.

3.2.2 Fabrication-Time Trojan Implementations

There are three types of hardware Trojans a malicious foundry can craft into an otherwise trusted IC layout: *additive*, *substitution*, and *subtractive*. Additive Trojans involve inserting additional circuit components and/or wiring into an existing design. Substitution Trojans require removing logic with low observability to make room for additional Trojan circuit components and/or wiring in an existing circuit design. Lastly, subtractive Trojans require removing circuit components and/or wiring to alter the behavior of a existing circuit design. The focus of this chapter is **estimating the susceptibility of a circuit layout**

to *additive* Trojan attacks. Substitution and subtractive Trojans, while intriguing, remain largely unexplored by the community. I do not know of any demonstrably stealthy and controllable substitution or subtractive Trojans and when researchers do create such an attack, there exists orthogonal mitigation strategies [190].

Inserting an additive Trojan at an untrusted foundry requires modifying two fundamental characteristics of an IC's physical layout—placement and routing—regardless of how an attacker implements the Trojan's trigger and payload. I define ***Trojan placement*** to be the act of placing additional hardware components into an IC layout for the purpose of crafting a Trojan trigger and payload, ***Victim/Trojan integration*** to be wiring the Trojan's payload to, or in the vicinity, of a security-critical net in the victim IC layout, and ***intra-Trojan routing*** to be the act of wiring the hardware Trojan together. The most challenging aspect of inserting a hardware Trojan at fabrication-time is finding empty space on the IC's device layer to insert the trigger and payload components (**Trojan placement**), AND routing the payload to a security-critical net (**Victim/Trojan integration**). ICAS estimates each of these fundamental tasks, in turn identifying weak points in the IC layout that an attacker might exploit.

3.3 Threat Model

In this chapter, I adopt the threat model for foundry-side attacks—§2.3.1 and Figure 2.3A—that assumes all steps in the IC design process can be trusted, *except* for all of the processes performed by a foundry, and its sub-contractors. I further restrict this threat model to fabrication-time attacks involving *additive* Trojans, i.e., hardware Trojans that require inserting additional circuitry into a physical IC design. Previous work on substitution/subtractive hardware Trojans shows that such Trojan insertion methods are addressable by measuring the controllability and observability of logic at the *behavioral* and/or *structural* level of the IC design, for which several methods have already been proposed [51, 58, 137, 138, 184, 202]. Orthogonally, this work fills the void of quantifying the

susceptibility of an IC design to additive hardware Trojan insertion at the *physical* level of the IC design process by an untrusted foundry.

Focusing on additive hardware Trojans, an adversary can only insert additional components/wires. They cannot increase the size of the chip to make additional room for the implants because this is readily caught by defenders. As a result, an attacker has two choices: *find* open space in the design large enough to accommodate the additional circuitry, or *create* open space in the design by moving circuitry around. The latter is extremely challenging due to its recursive nature, it runs the risk of violating fragile timing constraints and manufacturing design rules, and it increases fabrication turnaround time (which is usually set to three months); any of which could expose the Trojan. Therefore, my focus is identifying open spaces suitable for hardware Trojan implementation.

3.4 Untrusted Foundry Defenses

To protect IC layouts against insertion of a hardware Trojan by attackers at an untrusted foundry, two classes of defenses exist: **undirected** and **directed**. Undirected defenses leverage existing tuning knobs available during the IC layout process, but do not differentiate between security-critical and general-purpose wires and logic. Thus, undirected approaches provide probabilistic protection. On the other hand, directed defenses require augmenting existing PaR tool flows to harden the resulting IC layout, focusing on deploying defenses systematically around security-critical wires and logic. Thus directed approaches provide targeted protection, but increase the complexity of the place-and-route process.

This section provides an overview of the landscape of undirected and directed defenses. The focus is the mechanism each defense uses to increase the complexity faced by a foundry-level attacker. I use the results of the defensive analysis in this section to develop a set of unifying coverage metrics in the next section. Finally, in the evaluation, I evaluate commercial IC layouts using the defense-inspired metrics to quantify each defense's coverage.

3.4.1 Undirected

The lowest cost approach for protecting an IC layout from a foundry-level attacker is to take advantage of existing physical layout parameters (e.g., core density, clock frequency, and max transition time) offered by commercial CAD tools [195]. The goal is to increase congestion across the component layer and the routing layer. Ideally, this also results in increased congestion around security-critical logic and wires. Practically, increases in congestion around security-critical logic and wires is probabilistic.

Increased congestion is a symptom of increased resource utilization; hence, there are fewer resources available to the attacker. The most obvious resource that an attacker cares about are placement sites on the component layer. Increasing the density, *decreases unused placement sites*. Without sufficient placement sites, the attacker cannot implement their Trojan logic. A less obvious resource is attachment points on security-critical wires that serve as victim/Trojan integration points. Increasing routing layer congestion (via density and/or timing constraints) *increases the blockage around security-critical wires*, meaning there are less integration points.

3.4.2 Directed

To address the shortcoming of undirected approaches, recent defenses advocate focusing on security-critical logic and wires. Specifically, the approaches aim to prevent the attacker from being able to implement their hardware Trojan by occupying unused placement sites (i.e., transistors) [9, 10]. The challenge is that the filler cells used by these defenses must be tamper-evident, i.e., a defender must be able to detect if an attacker removed filler cells to implement their Trojan. Previous work shows that filling the entire component layer with tamper-evident filler cells (e.g. [195]) is infeasible due to routing congestion [10]. To make routing feasible, the most recent placement-centric defense focuses on filling the unused placement sites nearest security-critical logic first [9, 10].

Such placement-centric defenses increase the complexity faced by the attacker in two

ways. First, it is harder for the attacker to find *contiguous unused placement sites* to implement their Trojan's logic. Second, an indirect complication is increased *intra-Trojan routing* complexity. The more distributed the attacker's placement sites, the more long (i.e., uses upper routing layers) routes the attacker must create. Additionally, since the unused placement sites are far away from security critical logic, the attacker must make a longer, more complex, route to connect their hardware Trojan to the victim security-critical wire.

3.5 Unified Attack Metrics

Drawing from existing untrusted foundry defenses, I create an extensible set of IC layout attack metrics. I unify the objectives of existing defenses by decomposing the act of inserting a hardware Trojan into ICs at an untrusted foundry into three fundamental tasks and corresponding metrics:

1. Trojan logic placement: **Trigger Space**
2. Victim/Trojan integration: **Net Blockage**
3. Intra-Trojan routing: **Route Distance**

These tasks and accompanying metrics are the foundation for our methodology of assessing defensive coverage of an IC layout against an untrusted foundry. I implement our methodology as ICAS.

3.5.1 Challenges of Trojan Placement

The first phase of mounting a fabrication-time attack is Trojan placement. This requires locating unused placement sites on the placement grid to insert additional circuit components. While prior work [9, 10, 195] employs the notion of limiting the quantity of unused placement sites as a defense against fabrication-time attacks, how can I characterize unused

placement sites to gain insight into the feasibility of a fabrication-time attack on a given IC layout?

Only 60–70% of the placement cites are occupied in a typical IC layout to allow space for routing [196]. To facilitate intra-Trojan routing, an attacker prefers open placement sites form contiguous (adjacent) regions. This allows the attacker to drop-in a pre-designed Trojan, or if one had not been pre-designed, it minimizes the intra-Trojan routing complexity by confining the intra-Trojan routing to the lowest routing layers, i.e., reducing the jumping and jogging of nets. Such adjacency is classified in image processing as "4-connected". Therefore, a key factor that determines the difficulty of mounting fabrication-time attacks is the difficulty of inserting additional circuit components into a finalized IC design. I rank this difficulty in increasing order as follows.

1. **Trivial:** the Trojan components fit within a single contiguous group of 4-connected placement sites.

2. **Difficult:** the Trojan components must be split across multiple contiguous groups of 4-connected placement sites. The more placement site groups required, the more difficult intra-Trojan routing becomes.

3. **Not Possible:** the total area required by the hardware Trojan exceeds that of available placement sites.

Figure 3.2 illustrates these difficulty levels. The susceptibility of an IC design to fabrication-time attack can therefore be partially quantified by the size and number of contiguous open sites on the placement grid. This is the basis for ICAS' *Trigger Space* metric.

3.5.2 Challenges of Victim/Trojan Integration

Routing the Trojan payload to the targeted security-critical net requires the attacker to locate the nets of interest in the IC layout. I assume the worst case: the attacker has knowledge of all security-critical nets in the design, particularly, the nets they are trying to extract

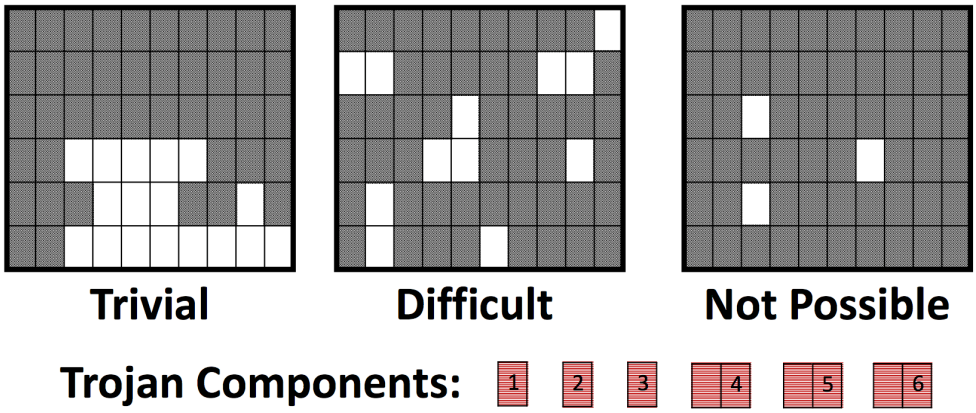

Figure 3.2: **Trojan Placement Difficulty.** Assume an attacker is attempting to insert 6 additional Trojan components that consume a total of 9 placement sites (as shown). If inserting these components on the *Trivial* placement grid (left), they can be placed adjacent to each other to simplify intra-Trojan routing. If inserting these components on the *Difficult* placement grid (middle), they must be scattered across the grid, making intra-Trojan routing more challenging. The *Not Possible* placement grid (right) does not have enough empty placement sites to accommodate the Trojan components.

information from or influence. An example of such a net in the OR1200 processor [121] is the net that holds the privilege bit. The attacker can acquire this knowledge either through a design-phase co-conspirator or through advanced reverse-engineering techniques [196]. No matter how the attacker gains this information, I assume they have it with zero additional effort.

I extend this threat to include nets that influence security-critical nets. To increase stealth, an attacker could also trace backwards from the targeted security-critical net, through logic gates, to identify nets that influence the value of the targeted security-critical net. This is called the *fan-in* of the targeted net. By connecting in this way, the attacker sacrifices controllability for stealth as their circuit modification is now physically separated from the security-critical net. To gain back controllability, attackers must create a more complex (hence larger) trigger circuit—decreasing the Trigger Space score, as well as increasing the likelihood of visual and/or side-channel detection. This trade-off limits how many levels back the attacker can integrate their payload.

No matter if the attacker is attacking the targeted security-critical wire directly or indi-

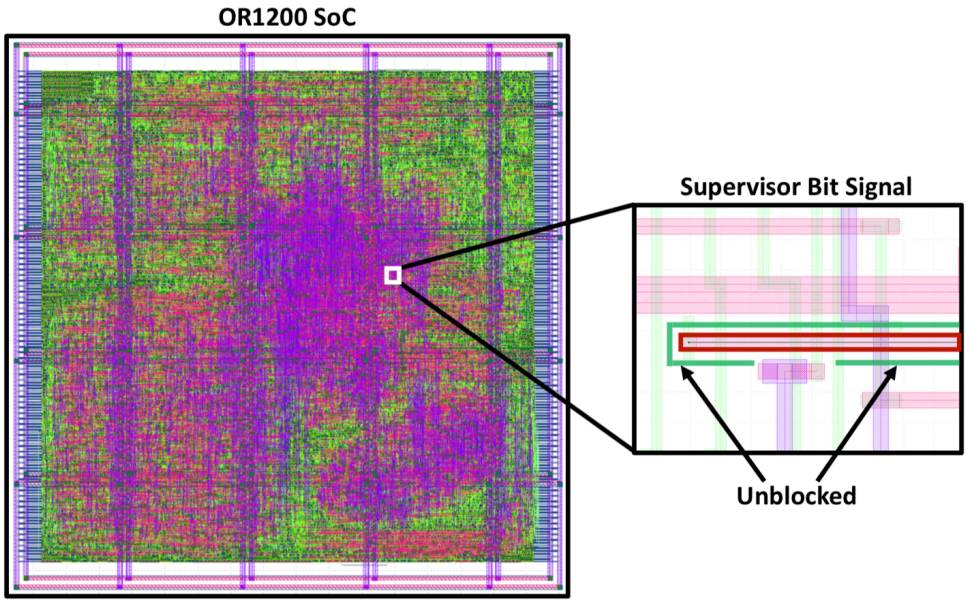

Figure 3.3: **Challenges of Victim/Trojan Integration.** The supervisor bit signal of the OR1200 processor SoC is the data input to the supervisor register of the OR1200 CPU. The supervisor register stores the privilege mode the processor is currently executing in. Changing the value on this net changes the privilege level of the processor allowing an attacker to execute privileged instructions. The more congested the area around this net, the more difficult it is for a foundry-level attacker to attach (or route in close proximity) a rogue wire to it.

rectly, the attacker must attach to some victim wire or route directly adjacent to it. Since an IC layout is three-dimensional, it is possible for the attacker to attach to any open point on the victim wire, either on the same layer (i.e., North, South, East, West) or from an adjacent layer (i.e., above or below). In the worst case, there are no other nets blocking the attacker from attaching to the targeted security-critical net or its N-level-deep influencers. In the best case, all attachment points are blocked by other nets. To quantify the number of points along, above, and below a targeted security-critical wire—and its N-deep fan-in—I implement the *Net Blockage* metric. Figure 3.3 shows the open (unblocked) integration points for the privilege net on the OR1200 processor.

3.5.3 Challenges of Intra-Trojan Routing

The final phase of a fabrication-time attack is Intra-Trojan routing. Intra-Trojan routing requires connecting the components that comprise the trigger and payload portions of the hardware Trojan together—including connecting to the integration point with the victim—to form a complete hardware Trojan. In the worst case, the attacker is able to find a single contiguous region to place the trigger and payload components that is nearby the victim security-critical net. Thus, routing the trigger and payload components will be trivial and the wire used to inject the payload will be short. In the best case, the attacker will have to implement their attack using many 4-connected placement regions (i.e., low Trigger Space score) and the only integration point on the targeted security-critical net (i.e., high Net Blockage score) is as far away from the open placement regions. Hence, I focus on quantifying the difficulty of routing the payload output to open attachment points on targeted security-critical nets (and its N-deep fan-in). To this end, I identify two challenges of intra-Trojan routing:

- Comply with design and fabrication rules
- Meet Trojan and payload-delivery timing requirements

3.5.3.1 Complying with Design Rules

For each process technology, there are many rules associated with how wires and components must be laid out in a design. Some of these rules are defined in the Library Exchange Format (LEF) [29] and contained in files that are loaded by modern Computer Aided Design (CAD) tools throughout the IC design process. There are two types of design rules: 1) those regarding the construction of circuit components (i.e., standard cells), and 2) those regarding routing. I classify these as *component design rules* and *routing design rules*, respectively. As technology nodes shrink, both rule sets are becoming increasingly complex [147].

It is vital for an attacker to comply with these design rules as violating them risks exposure. If an attacker inserts additional logic gates (standard cells) by making copies of existing components in a design, they can avoid violating *component design rules* involved with Trojan placement. However, to connect a wire from the Trojan payload to security-critical target net(s), they must perform custom Trojan routing. Therefore, complying with *routing design rules* is a concern. Routing design rules include specifications for the minimum distance between two nets on a specific routing layer, the minimum width of nets on a given layer, etc. Complying with these rules becomes easier for an attacker if security-critical net(s) are not blocked by other wires or components. The higher the Net Blockage score, the more difficult it is to make a connection, the more complex—and error prone—the route.

3.5.3.2 Meeting Timing Requirements

Every wire in an IC has a resistance and a capacitance, making it behave like an RC circuit, i.e., there is a time delay associated with driving the wire *high* (logic 1) or *low* (logic 0). The longer the wire, the more time delay there is [45]. If the security-critical net(s) has timing constraints (e.g., setup and hold times) that dictate when the payload signal must arrive for the attack to be successful, the Trojan routing must meet these constraints. Furthermore, the farther the security-critical net is from the payload circuit, the more obstacles that must be routed around, increasing the routing distance even further. This is the basis for ICAS' *Route Distance* metric. A natural limit for Route Distance is dictated by the clock frequency of the victim circuit, as most attacks must operate synchronously with their victim.

3.6 Extensible Coverage Assessment Framework

The ICAS framework is comprised of two tools, **Nemo** and **GDSII-Score**, as shown in Figure 3.4. Nemo identifies security-critical wires based on designer annotations and circuit

dataflow, while GDSII-Score assesses the defensive coverage of a given IC layout against a set of attacks. ICAS takes as input four sets of files: 1) gate-level netlist (generated *after* all physical layout optimizations), 2) process technology files, 3) physical layout files, and 4) set of attacks. The process technology files include a Library Exchange Format (LEF) file and layer map file [27, 29]. The physical layout files include a Design Exchange Format (DEF) file and the GDSII file of an IC layout [29, 31]. The attack files are are a list of properties for each attack to assess coverage against: number of transistors, security-critical wire(s) to attach to, and timing constraints. All ICAS input files except the attack files are either generated-by or inputs-to the back-end IC design phase, and hence are readily available to back-end designers.

Though ICAS is extensible, our implementation includes three security metrics that capture the challenges faced by a foundry-level attacker looking to insert a hardware Trojan: amount and size of open-placement regions (Trigger Space), quantity of viable attachment points to targeted security-critical (and influencer) nets (Net Blockage), and the proximity of open placement regions to targeted security-critical net(s) (Route Distance). Together with the attack requirements, these metrics quantify the complexity an attacker faces for each step of inserting specific hardware Trojans into the given IC layout. I describe the implementation of both ICAS components below.

3.6.1 Nemo

Nemo is the first analysis tool in the ICAS framework. It bridges the semantic gap between the human readable RTL netlist and post-PaR netlist. Additionally, Nemo broadens the set of "security-critical" nets by performing a fan-in analysis of root security-critical nets. This is necessary since the inter-connected nature of signals within a circuit design means an adversary could influence the state of security-critical nets by controlling a net that is a part of its fan-in. Nemo takes as input a Verilog netlist and automatically identifies security-critical nets in the post-PaR netlist HDL, which it outputs in the form of a

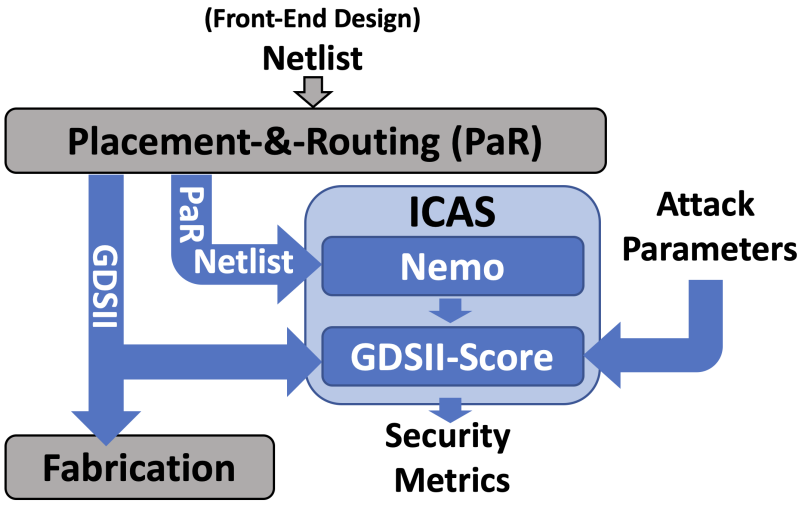

Figure 3.4: **ICAS Work Flow.** ICAS consists of two tools, **Nemo** and **GDSII-Score**, and fits into the existing IC design process (Fig. 2.1) between PaR and fabrication. Nemo analyzes a gate-level (PaR) netlist and traces the fan-in to security-critical nets in a design. GDSII-Score analyzes a GDSII file (i.e., an IC layout) and computes metrics quantifying its vulnerability to a set of foundry-level attacks.

Graphviz dot file. Similar to prior work [74, 102, 205], Nemo assumes that a unique signal name prefix (within the RTL HDL) has been appended to various signals considered "security-critical". I make this assumption since determining what signals are "security critical" requires contextual knowledge of how the design will be used.

3.6.1.1 Annotating Security-Critical Signals in the RTL Netlist

The process of uncovering and annotating security-critical signals in the RTL netlist is Security-Critical Component Identification (SCCI). While SCCI is an active area of research in the hardware security community, orthogonal to addressing the untrusted foundry problem, there are two approaches I are aware of: *manual* and *semi-autonomous* identification. The first, and most traditional, is *manual identification*. Manual identification requires a human expert to study the design's specification (e.g., Instruction Set Architecture in the case of a processor), and identify properties that are critical to the security of software or other hardware utilizing the design [59, 74]. The second, and current state-of-the-art developed by Zhang *et al.* [205], is *semi-autonomous identification*. *Semi-autonomous*

identification involves two steps. First, a program observes a variety of test-benches exercising the design to generate a large set of possible invariants defined over the hardware specification. Second, a pre-trained penalized logistic regression classifier is used to classify which invariants, or portions of the specification, are security-critical. This method of SCCI is *semi-autonomous*, as it requires the classifier model be pre-trained with either existing published errata on previous versions of the hardware design, or using *manual identification*. While I perform manual SCCI, results reported by Zhang *et al.* [205] suggest that their tool would result in a similar set of root security-critical signals.

3.6.1.2 Identifying Security-Critical Signals in the PaR Netlist

While there are existing (aforementioned) techniques for identifying and annotating security-critical components in the RTL netlist, unfortunately, these techniques do not track security-critical signals past the RTL design phase and do not capture data-flow. Thus, Nemo's core task is to bridge the semantic gap and uncover duplicated or renamed security-critical signals in the post-PaR netlist. Fortunately, while synthesis and layout tools do modify a netlist by duplicating and removing signals and components (as part of optimization and meeting performance requirements), they do not completely rename existing signals. This makes it possible for Nemo to identify root security-critical signals (flagged at the behavioral level) by name at the physical level. ***To avoid removal of security-critical signals, I modify synthesis and layout scripts to essentially lock them in place.*** Nemo works backwards from root security-critical signals to identify the fan-in to these signals. The search depth is a configurable parameter of Nemo.

3.6.1.3 Implementation

Nemo is implemented as a back-end target module to the open-source Icarus Verilog (IVL) [192] Verilog compiler and simulation tool written in C++. The IVL front-end exposes an API to allow third-parties to develop custom back-end target modules. Nemo is a

custom target module (also written in C++) designed to be loaded by IVL. Since gate-level netlists are often described with the same HDL that was synthesized to generate the netlist (e.g., Verilog), I utilize the IVL front-end to interpret the Verilog representation of the netlist and our custom back-end target module, Nemo, to perform a breadth-first search of the post-PaR netlist. I open-source Nemo [111] and release instructions on how to compile and integrate Nemo with IVL.

3.6.2 GDSII-Score

GDSII-Score is the second analysis tool in the ICAS framework. GDSII-Score is an extensible Python framework for computing security metrics of a physical IC layout. It takes as input the following: Nemo output, GDSII file, DEF file, technology files (LEF and layer-map files), and attacks description file. First, GDSII-Score loads all input files and locates the security-critical nets within the physical layout. Next, it computes security metrics characterizing the susceptibility of an IC design to each of the input attacks. Specifically, the three security metrics that I implement are: 1) **Trigger Space**: the difficulty of implementing the hardware Trojan, 2) **Net Blockage**: the difficulty of Trojan/victim integration, and 3) **Route Distance**: the difficulty of meeting Trojan timing constraints. I open source the GDSII-Score framework and our security metric implementations [110].

3.6.2.1 Metric 1: Trigger Space

The Trigger Space metric estimates the challenges of Trojan placement (§3.5.1). It computes a histogram of open 4-connected regions of all sizes on an IC's placement grid. The more large 4-connected open placement regions available, the easier it is for an attacker to locate a space to insert additional Trojan circuit components at fabrication time. A placement site is considered to be "open" if the site is empty, or if it is occupied by a filler cell. Filler cells, or capacitor cells, are inserted into empty spaces during the last phase of layout to aid fabrication. Since they are inactive, an attacker can create empty placement

sites by removing them, without altering the functionality or timing characteristics of the victim IC.

To compute the trigger space histogram, GDSII-Score first constructs a bitmap representing the placement grid. Placement sites occupied by standard cells (e.g., NAND gate transistors) are colored while those that are open are not. Information about the size of the placement grid and the occupancy of each site in the grid is available in the Design Exchange Format (DEF) file produced by commercial PaR tools. GDSII-Score then employs a breadth-first search algorithm to enumerate the maximum size of all 4-connected open placement regions.

3.6.2.2 Metric 2: Net Blockage

The Net Blockage metric estimates the challenges of integrating the hardware Trojan's payload into the victim circuit (§3.5.2). It computes the percent blockage around security-critical nets and their influencers. The more congested the area surrounding security-critical nets, the more difficult it is to attach the Trojan circuitry to these nets. There are two types of net blockage that are calculated for each security-critical net: *same-layer* and *adjacent-layer*.

Same-layer blockage is computed by traversing points around the perimeter (North, South, East, West) at a granularity of g, at a specific distance, d, around the security-critical net and determining which points lie within other circuit components, as detailed in Figure 3.5a. To determine if a specific point along the perimeter lies within the bounds of another circuit component, I utilize the point-in-polygon ray-casting algorithm [66]. The extension distance, d, around the security-critical path element and the granularity of the perimeter traversal, g, are configurable in our implementation. However, I default to an extension distance of one wire-pitch and a granularity of 1 database units, respectively, as defined in the process technology's LEF file. The IC designs used in our evaluation are built using a 45 *nm* process technology, for which 1 database units is equivalent to 0.5 *nm*.

Additionally, an open region is considered "blocked" if it is not wide enough for a minimal width wire to be routed through while maintaining the minimal amount of wire spacing required on that metal layer, as defined in the LEF file. The percentage of the perimeter length that is blocked by other circuit components is considered the same-layer blockage percentage.

Adjacent-layer blockage is computed by analyzing the area directly above and below a security-critical net, and computing the total area of overlap between other components, as detailed in Figure 3.5b. To calculate this overlap area I utilize an overlapping sliding window approach. Additionally, any un-blocked regions above or below the security-critical net are considered "blocked" if they are not large enough to accommodate the smallest possible via geometry allowed on the respective via layer, as defined in the LEF file. The percentage of the total top and bottom area that is blocked by nearby circuit components is the adjacent-layer blockage percentage.

The same-layer and adjacent-layer blockage percentages are combined via a weighted average to form a comprehensive *overall* net blockage percentage where 66% is based on same-layer blockage (north, south, east, and west) and 33% is based on adjacent-layer blockage (top and bottom). I weight the same-layer blockage by 66%, or $\frac{2}{3}$, because 4 out of 6 total sides of a wire (**north**, **south**, **east**, **west**, top, and bottom) are on the same layer. Likewise, I weight the adjacent-layer blockage by 33%, or $\frac{1}{3}$.

Lastly, a total *same-layer*, *adjacent-layer*, and *overall* net blockage metric is computed for the entire IC design. For an IC design with n security-critical nets, the *same-layer* (b_{same}), *adjacent-layer* (b_{adjacent}), and *overall* (b_{overall}) net blockage metrics are computed according to equations 3.1, 3.2, and 3.3, respectively.

$$b_{\text{same}} = \frac{\sum_{i=1}^{n} perimeter\_blocked_n}{\sum_{i=1}^{n} perimeter_n} \tag{3.1}$$

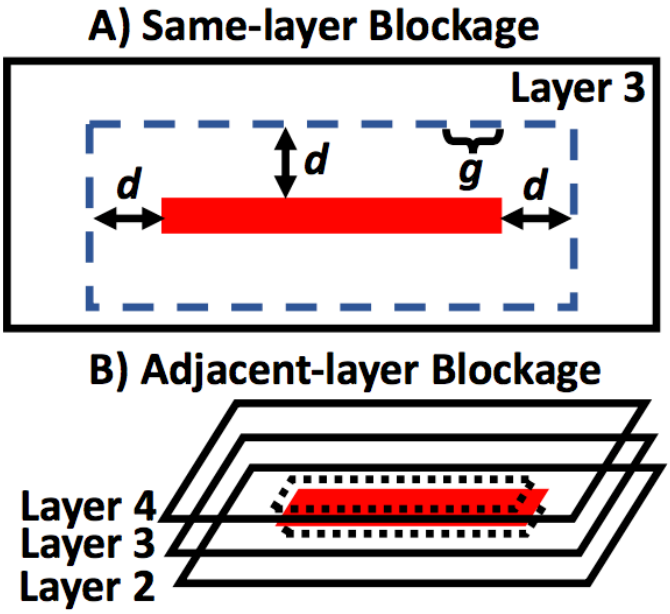

Figure 3.5: **Net Blockage Algorithm.** A) Same-layer net blockage is computed by traversing the perimeter of the security-critical net, with granularity g, and extension distance d, and determining if such points lie inside another component in the layout. B) Adjacent-layer net blockage is computed by projecting the area of the security-critical net to the layers above and below and determining the area of the projections that are occupied by other components.

$$b_{\text{adjacent}} = \frac{\sum_{i=1}^{n} area\_blocked_n}{\sum_{i=1}^{n} 2 * area_n} \tag{3.2}$$

$$b_{\text{overall}} = \left(\frac{2}{3} * b_{\text{same}}\right) + \left(\frac{1}{3} * b_{\text{adjacent}}\right) \tag{3.3}$$

3.6.2.3 Metric 3: Route Distance

The Route Distance metric combines the Net blockage and Trigger Space metrics (thus is correlated with these metrics) to estimate the difficulty of of meeting Trojan and attack timing constraints (§3.5.3). It computes a conservative estimate, i.e., Manhattan distance, for the minimal routing distance between open trigger placement sites and the n least blocked integration sites on the targeted security critical nets. It cross-references each Manhattan distance with the distribution of net lengths within the entire IC design. Net

length can impact whether or not the Trojan circuit will meet timing constraints and function properly. Understanding where in the distribution of net lengths the Trojan routing falls provides insights into the ability of the Trojan circuit to meet its timing requirements and is an opportunity for outlier-based defenses. In summary, the more Manhattan distances that fall within one standard deviation of the mean net length, the easier it is to carry out an attack.

I implement the Route Distance metric as follows. First, the Net Blockage and Trigger Space metrics are computed. Next, the the distribution of all net-lengths within the IC layout are computed. Then, two-dimensional Manhattan distances between all unblocked nets ($<$ 100% overall net blockage) and trigger spaces are calculated. The Manhattan distance calculated is the minimum distance between a given trigger space and security-critical net, i.e., the minimum distance between any placement site within the given trigger space and any unblocked location on the targeted security-critical net. Lastly, each Manhattan distance is reported in terms of standard deviations away from the mean net-length in the given IC layout.

3.7 Evaluation

I use ICAS to quantify the defensive coverage of existing defensive layout techniques—revealing that gaps persist. First, I analyze the effectiveness of undirected defenses [195]. Specifically, I measure the impact of varying both physical and electrical back-end design parameters of the same IC layout on its susceptibility to attack. Second, I analyze the effectiveness of directed defenses [9, 10]. Specifically, I measure the coverage of existing, placement-oriented, defensive layout schemes in preventing the insertion of an attack by the foundry. Beyond revealing gaps, our results reveal that there is an opportunity for improving both directed and undirected defenses that systematically eliminates Trojan/victim integration points. Lastly, our evaluation also demonstrates that ICAS is design-agnostic, works with commercial tools, and scales to complex IC layouts.

Table 3.1: Hardware Trojans used in defensive coverage assessment.

Trojan	# Std Cells	# Placement Sites	Timing Critical?
A2 Analog [196]	2	20	✗
A2 Digital [196]	91	1444	✓
Privilege Escalation [58, 85]	25	342	✓
Key Leak [136]	187	2553	✓

3.7.1 Experimental Setup

I utilize three IC designs for our evaluations: *OR1200 processor SoC*, *AES accelerator*, and *DSP accelerator*. The OR1200 processor SoC is an open-source design [121] used in previous fabrication-time attack studies [196]. The AES and DSP accelerator designs are open-sourced under the Common Evaluation Platform (CEP) benchmark suite [108]. The OR1200 processor SoC consists of a 5-stage pipelined OR1200 CPU that implements the 32-bit OR1K instruction set and Wishbone bus interface. The AES accelerator supports 128-bit key sizes. The DSP accelerator implements a Fast Fourier Transform (FFT) algorithm.

All designs target a 45*nm* Silicon-On-Insulator (SOI) process technology. I synthesize and place-and-route all designs with Cadence Genus (version 16.23) and Innovus (version 17.1), respectively. In our first evaluation (§3.7.2) the design constraints (clock frequency, max transition time, core density) used for both synthesis and layout are varied as noted. However, in our second evaluation (§3.7.3) the same design constraints (100 MHz clock frequency, 100 *ps* max transition time, 60% core density) were used for both synthesis and layout to form a common baseline. All ICs are synthesized and placed-and-routed on a server with 2.5 GHz Intel Xeon E5-2640 CPU and 64 *GB* of memory running Red Hat Enterprise Linux (version 6.9).

3.7.1.1 Security-critical Signals

The first tool in the ICAS flow is Nemo. Nemo tracks security-critical signals from the HDL level to the IC layout level. The first step is flagging root security-critical signals at the RTL level, for each IC design. For the OR1200 processor SoC, the supervisor bit signal *supv* is flagged. I select this signal because one can alter the state of this bit to escalate the privilege mode of the processor [196]. For the AES accelerator, I flag all 128 key bits as security-critical. The *next_out* signal within the DSP accelerator was flagged as security-critical. The *next_out* signal of the DSP accelerator indicates to external hardware when an FFT computation is ready at the output registers. Tampering with the *next_out* signal allows the attacker to hide specific outputs of the DSP accelerator. Lastly, Nemo marks, for each design's IC layout, all root security-critical nets and their 2-deep fan-in as security-critical nets.

3.7.1.2 Hardware Trojans

Table 3.1 lists the hardware Trojan designs that I use in our evaluation. The first two Trojan designs (analog and digital variants of A2) are attacks on the OR1200 processor and DSP accelerator ICs. With respect to the OR1200, the A2 attacks act as a hardware foothold [85] for a software-level privilege escalation attack. With respect to the DSP accelerator, the A2 attacks suppress the *next_out* signal (§3.7.1). The Privilege Escalation Trojan targets solely the OR1200 and the Key Leak solely the AES accelerator.

3.7.1.3 Build Environment

Both ICAS tools (Nemo and GDSII-Score) were run on the same server as the synthesis and place-and-route CAD tools. Nemo and Icarus Verilog were compiled from source using GCC (version 4.4.7). For increased performance, GDSII-Score was executed using the PyPy Python interpreter with JIT compiler (version 4.0.1).

3.7.2 Undirected Defense Coverage

As detailed in §3.4.1, a defensive strategy for protecting an IC layout from foundry-level attackers is to exploit physical layout parameters (e.g., core density, clock frequency, and max transition time) offered by commercial CAD tools to increase congestion—hopefully around security-critical wires. The tradeoff is that while this is a low cost defense in that CAD tools already expose such knobs, the entire design is impacted and there is no guarantee that security-critical wires will be protected. I use ICAS and its three security metrics to quantify the effectiveness of such undirected approaches [195].

To uncover the impact of each parameter, I start by generating 60 different physical layouts of the OR1200 processor design by varying:

1. **Target Core Density** (%): 50, 70, 90
2. **Clock Frequency** (MHz): 100, 200, 500, 1000
3. **Max Transition Time** (ps): 100, 150, 200, 250, 300

Target core density is a measure of how congested the placement grid is. Typically, designers select die dimensions that achieve ∼60–70% placement density to allow space for routing [196]. Target clock frequency is the desired speed at which the circuitry should perform. Typically, designers select the clock frequency based on performance goals. Max transition time is the longest time required for the driving pin of a net to change logical values. Typically, designers choose a value for max transition time based upon power consumption and combinational logic delay constraints.

For each of the 60 layout variations I compute ICAS metrics. Figures 3.6, 3.7, and 3.8 provide a visual representation for each metric. Overlaid on Figure 3.8 are the number of unique attack (color-coded) implementations for each Trojan (Tab. 3.1) at six parameter configurations. Across the 60 IC layouts, the time it took ICAS to complete its analyses ranged from 38 seconds to 18 minutes. On average, this translates to less than 10% of the

combined synthesize and place-and-route run-times. These run-time results demonstrate the deployability of ICAS as a back-end design analysis tool. Overall, our evaluation indicates that while some of these layout parameters do increase attacker complexity, none are sufficient on their own. I break down the results metric-by-metric.

3.7.2.1 Trigger Space Analysis

Figure 3.6 shows the distributions of open trigger spaces across 15 unique OR1200 layouts. I vary target core density and max transition time parameters across layouts, while I fix the target clock frequency at 1 GHz. A *trigger space* is defined as a contiguous region of open placement sites on the device layer placement grid and is measured by number of contiguous "4-connected" placement sites. Each box represents the middle 50%, or interquartile range (IQR), of open trigger space sizes for a given IC layout. The dots represent individual data points within and outside the IQR. Our empirical results affirm prior notions [9, 10, 195] that increasing the target core density of an IC layout results in fewer large open spaces to insert hardware Trojans. Additionally our results indicate that at lower densities, decreasing the max transition time constraint decreases the median trigger space size. Similar trends occur at lower clock frequencies. While results show that modulating target core density is effective, observe that even in the best case, large trigger spaces remain.

From our Trigger Space analysis, I conclude future undirected defenses should modulate layout parameters that both 1) shrink the trigger space IQR, and 2) shift the median towards one. In doing so, defenders: 1) minimize the variation in sizes of contiguous open-spaces available to the attacker—therefore limiting their Trojan design (size) options, and 2) force the attacker to have to distribute Trojan components across the die making *Trojan logic placement* and *intra-Trojan routing* more challenging.

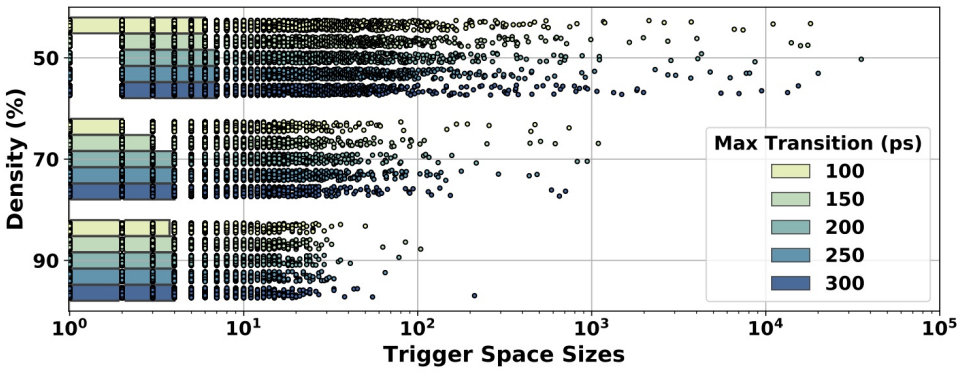

Figure 3.6: **Trigger Space Results.** Trigger Space distributions for 15 different OR1200 processor IC layouts. Core density and max transition time parameters are varied across the layouts, while target clock frequency is held constant at 1 GHz. The boxes represent the middle 50% (interquartile range or IQR) of open placement regions in a given layout, while the dots represent individual open placement region sizes.

3.7.2.2 Net Blockage Analysis

Figure 3.7 shows the Net Blockage metric (Eq. 3.3) computed across 20 unique OR1200 layouts. I fix the target density at 50% across all layouts, while the target clock frequency and max transition time are varied (as listed above). The results show that at lower clock frequencies a smaller max transition time parameter corresponds to increased Net Blockage. This corresponds to less open Trojan/victim integration points available to the attacker. However, as clock speed increases, the correlation between max transition time and overall Net Blockage deteriorates. Intuitively, smaller max transition times should lead to smaller average net-lengths within the design, as transition time is a function of the capacitive load on the net's driving pin [45]. Shorter net-lengths result in more routing congestion as components cannot be spread-out across the die. However, capacitive load (on a driving pin) is inversely proportional to frequency, thus at higher clock frequencies the max-transition time constraint is more easily satisfied, and altering it has less effect on the Net Blockage. Given these results, the effectiveness of modulating transition time is context dependent and—even in the best case—open integration points remain.

From our Net Blockage analysis, I conclude future undirected defenses should mod-

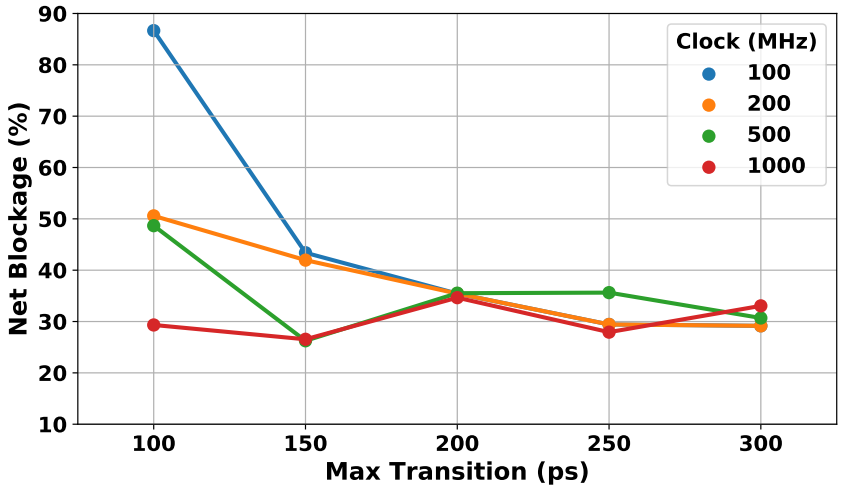

Figure 3.7: **Net Blockage Results.** *Overall* Net Blockage results computed across 20 different OR1200 processor IC layouts. A target density of 50% was used for all layouts, while target clock frequency and max transition time parameters were varied.

ulate layout parameters that both 1) shrink overall security-critical wire lengths, and 2) maximize routing congestion in the vicinity of security-critical wires. In doing so, defenders minimize the *Victim/Trojan integration* attack surface.

3.7.2.3 Route Distance Analysis

Figure 3.8 shows the Route Distances across six various OR1200 layouts in the form of heatmaps that capture the trade space between layout parameters. Core density and max transition times were varied across the layouts (indicated in the labels), while clock frequency was held constant at $100\,MHz$. Each heatmap describes several (column-wise) histograms of Route Distances in terms of standard deviations from the mean net length observed in that particular IC layout (y-axis). The Route Distances reported are those between any unblocked security-critical nets, and trigger spaces large enough to hold an attack of a given size range (x-axis). That is, the color intensities within in a given heatmap column indicate the percentage of (security-critical-net, trigger-space) pairs in that column that are within a range of distance apart. Additionally, overlaid on each heatmap are rectangles indicating the region of the heatmap where a given attack (Tab. 3.1) can be implemented,

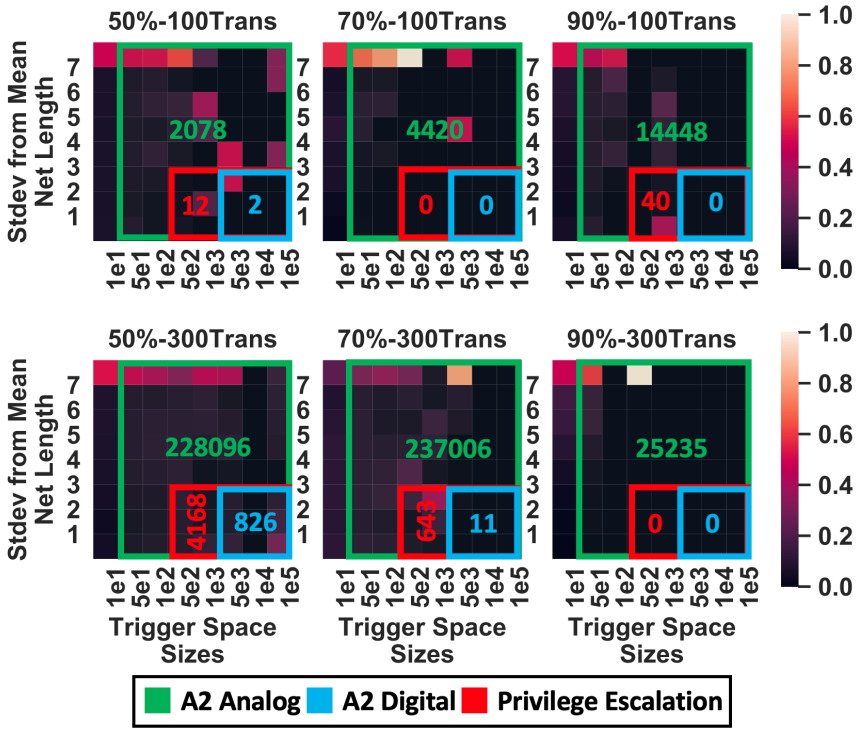

Figure 3.8: **Route Distance Results.** Heatmaps of routing distances across six unique IC layouts of the OR1200 processor. Core density and max transition times are labeled. Each heatmap is to be read column-wise, where each column is a histogram, i.e, the color intensity within a heatmap column indicates the percentage of (critical-net, trigger-space) pairs that are within a (y-axis) distance apart. Overlaid are rectangles, indicating regions on each heatmap a given attack can exploit, and numbers indicating the number of unique attack implementations.

and the number of possible attack configurations, (security-critical-net, trigger-space) pairs, that can be exploited.

If timing is critical to the operation of an attacker's desired Trojan, (critical-net, trigger-space) pairs with routing distances significantly greater than the average net length in the IC layout are less likely to be viable candidates for constructing hardware Trojans. IC layouts with few desirable (critical-net, trigger-space) pairs are much more time-consuming to attack. Namely, the IC layouts with heatmaps that indicate a higher percentages of far-apart (critical-net, trigger-space) pairs, where the trigger spaces are small, are most secure. From Figure 3.8, I conclude that at high density, max transition time has little affect on IC layout security; while at lower densities, lower max transition time designs are more secure. Similar trends exist across other layout parameters, as shown in Figures A.1–A.3

in Appendix A.

From our Route Distance analysis, I conclude future undirected defenses should modulate layout parameters that maximimze the distance between security critical wires and open trigger spaces. In doing so, defenders: 1) maximize intra-Trojan routing difficulty, and 2) restrict attackers from implanting timing-critical Trojans.

3.7.2.4 Cost of Varying Layout Parameters

The results indicate that increasing core density is effective, but incomplete, and increasing clock frequency and decreasing max transition time is marginally effective and incomplete. While tuning these parameters is low cost to the designer, there is a cost to the design in terms of complexity and power requirements. I elucidate by discussing how varying each design parameter (density, clock frequency, and max transition time) impacts non-security characteristics of a circuit design.

While increasing core density to 90% makes placing-and-routing a Trojan more difficult, it also makes placing-and-routing the rest of the design more challenging. Specifically, it can become nearly impossible to meet timing closure for the entire design if there is not enough space within the core area to re-size cells and/or add additional buffer cells. Depending on performance and security requirements, a layout engineer may choose to relax timing constraints in order to achieve a higher core density. Alternatively, a layout engineer may attempt to surround security-critical nets with areas of high densities, while maintaining a lower overall core density, as previously suggested [9, 10].

Decreasing the maximum transition time and increasing the clock speed of an entire circuit design makes it more difficult to place-and-route a functional Trojan that meets timing constraints, but also directly impacts the performance characteristics of the circuit. Additionally, it is important to note that max transition time is related to the clock frequency, so varying one without the other changes performance tolerances. While increasing the performance of the design might increase security, it comes at the cost of increasing power

consumption. Depending on the power-consumption requirements of the design, it may be possible for a designer to over-constrain these parameters for added security.

3.7.3 Directed Defense Coverage

As an alternative to probabilistically adding impediments to the attacker inserting a hardware Trojan, recent works proposes a directed approach. As detailed in §3.4.2, placement-centric directed defenses [9, 10] attempt to prevent the attacker from implementing their Trojan by occupying all open placement sites with tamper-evident filler cells. The limitation with such defenses is that it is infeasible to fill *all* open placement sites with tamper-evident logic [10]. Thus, the defenses focus their filling near security-critical logic, leaving gaps near the periphery of the IC layout. Whether these open placement sites near the periphery are sufficient to implement an attack is an open question.

The goal of this evaluation is to determine not only if it is still possible for a foundry-level attacker to insert a hardware Trojan, given placement-centric defenses, but to quantify the number of viable implementations available to the attacker—to act as a surrogate for attacker complexity. For the evaluation, I use our three IC designs (OR1200 processor SoC, AES accelerator, and DSP accelerator). For each design, I create two IC layouts: (1) unprotected and (2) protected. For the protected IC layout, I use the latest placement-centric defense [9]; using the identified security-critical wires (§3.7.1) to direct the defense. I lay out all IC designs using these parameters: target clock frequency of $100\,MHz$, max transition time of $100\,ps$, and a target core density of 60%.

I then use ICAS to asses the defensive coverage of each of the six IC layouts. This analysis has two goals: (1) determine whether the IC is vulnerable to attack and (2) understand the impact of applying the defense. I answer both questions in an attack-centric manner using the hardware Trojans in Table 3.1 to asses defensive coverage against. For each attack/IC layout combination I plot the number of (security-critical-net, trigger-space) pairs that could be used in implementing each Trojan. A (security-critical-net, trigger-space) pair

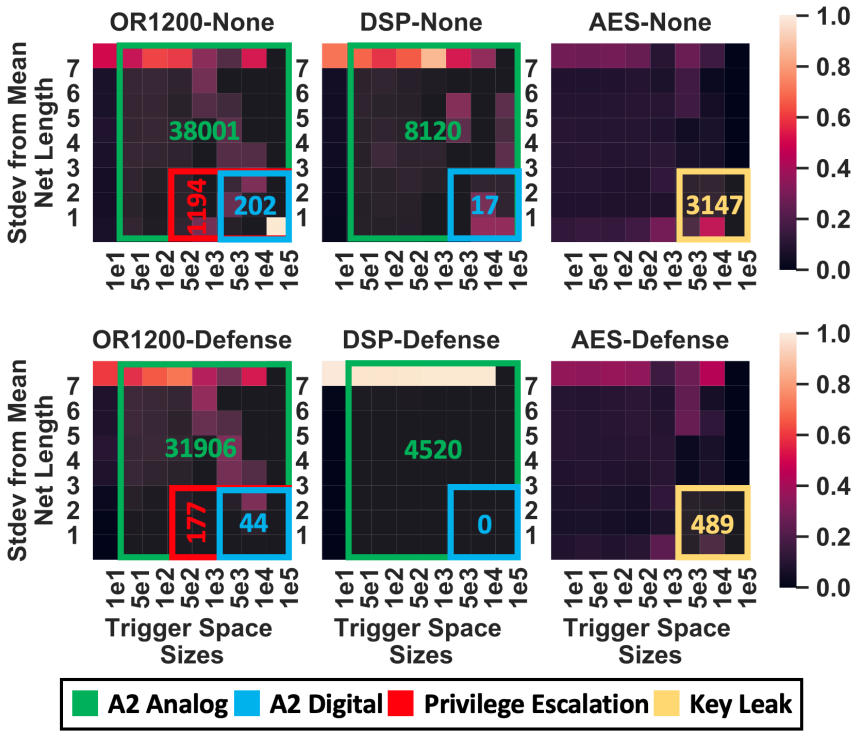

Figure 3.9: **Effectiveness of Layout-Level Defenses.** Routing Distance heatmaps across three IC designs, with and without the placement-centric defense described in [9, 10]. Heatmaps should be interpreted similar to Fig. 3.8.

is considered a viable candidate for implementing a Trojan if:

1. the trigger space size is at *least as large* as the minimum number of placement sites required to implement the desired hardware Trojan design

2. the security-critical net is less than 100% blocked

3. if the hardware Trojan is "Timing-Critical", i.e., it must function at the design's core operating frequency, then the distance between the trigger space and open integration point on the security-critical net must be ≤ 3 standard deviations from average net length; otherwise, any distance is allowed.[2]

Figure 3.9 shows the defensive coverage for each IC design. Overlaid on each heatmap

[2]Three standard deviations from the average net length is the threshold for Trojan-to-integration-point routing without violating timing constraints, because it accounts for 99.7% of the designs' wires—outliers tend to be power wires. For an exact calculation, it is possible to extract parasitics for a target Trojan's route to determine if it violates timing constraints.

are rectangles (and numbers) indicating unique possible attack implementations. These results show that existing placement-centric defenses are effective at *reducing* an IC's fabrication-time attack surface, compared to no defense—but *gaps persist*. Given that filling placement sites with tamper-evident logic is already maximized, these results point to systematically adding congestion around security-critical wires as a means to close all remaining defensive gaps; i.e., a directed version with similar effect to existing undirected defenses.

3.8 Discussion

ICAS is the first tool to provide insights into the security of physical IC layouts. It is extensible across many dimensions including CAD tools, process technologies, security metrics, and fabrication-time attacks and defenses. To demonstrate ICAS' capabilities I implemented three security metrics (net blockage, trigger space, and routing distance) using it. The focus of this paper is using these metrics to estimate the coverage of existing untrusted foundry defenses, which show that IC designs are still vulnerable to attack. I envision uses for ICAS beyond this, as an integral part of the IC design process using commercial tools.

3.8.1 ICAS-Driven Defensive Layout

ICAS provides an added notion of security to the IC layout (place-and-route) process to enable researchers to explore countermeasures against fabrication-time attacks. To the best of our knowledge, the existing targeted defensive IC layout techniques [9, 10, 195] are entirely *placement-centric*, i.e., filling unused space on the device layer with functional logic cells. While ICAS is capable of evaluating placement-centric defensive layout techniques, its security-insights also asses *routing-centric* defensive layout techniques. For example, layout engineers can leverage ICAS to create high degrees of routing congestivity in close proximity to security-critical nets. ICAS' security metrics enable IC layout designers to

optimize the security of both the placement *and* routing of their designs.

3.8.2 Constrained Security Metrics

In its primary state, ICAS focuses on computing metrics that reason about the *spatial* resources required to implant hardware Trojans in IC layouts. While our metrics are unconstrained and thus conservative, it is trivial to extend, and constrain, ICAS metrics to account for other layout resources that may impact an attacker's decision process. For example, even with a plethora of spatial resources available to insert Trojan components, doing so in certain areas of the chip may impact local power consumption enough to disrupt normal operating behavior. Alternatively, inserting a hardware Trojan nearby un-shielded, fast toggling, interconnects may negatively impact the Trojan's signal integrity, rendering it benign. I recognize it is impractical to consider all possible constraints, and hence I design ICAS to be extensible.

3.8.3 Extensibility of Security Metrics

GDSII-Score is the ICAS tool that computes security metrics from an IC layout. It loads several files describing the IC layout to instantiate a single Python class (called "Layout") that contains query-able data structures containing a polygon representation of all components in the layout. Additionally, GDSII-Score contains several subroutines that compute spatial relationships between polygon objects and points within the layout. From these data structures and the provided subroutines, it is trivial to integrate additional metrics into GDSII-Score. To facilitate additional metrics, I open-source GDSII-Score [110], and our three example metrics that demonstrate how to query the main "Layout" data structure.

3.8.4 Extensibility of CAD Tools

Almost all steps of the IC design process utilize CAD tools. ICAS integrates into a commercial IC design process after placement-and-routing (Figure 2.1). While ICAS is

validated with IC layouts generated by Cadence tools, integrating ICAS with other vendors' CAD tools does not require any additional effort due to the common process technology (LEF) and GDSII specifications used by ICAS.

3.8.5 Extensibility of Process Technologies

I test ICAS using IC layouts built with a 45 *nm* SOI process technology; however, ICAS is agnostic of process technology. The LEF and layer map files (§3.6) are the only ICAS input files that are dependent on the process technology. A LEF file describes the geometries and characteristics of each standard cell in the cell library, and the layer map file describes the layer name-to-number mappings, respectively, for a given process technology. ICAS adapts to different process technologies provided that all input files adhere to their specifications [27, 29].

3.8.6 Limitations

The goal of ICAS is to estimate the susceptibility of circuit layouts to *additive* hardware Trojans, thus there are limitations. First, as implemented, ICAS is not capable of estimating the susceptibility of a circuit layout to *subtractive* or *substitution* Trojans. I are unaware of any stealthy and controllable subtractive hardware Trojans, but should researchers develop such an attack, metrics will need to be added to ICAS to enable detection. Dopant-level Trojans are the closest example of substitution Trojans [17, 90]. Though their non-existent footprints make them difficult to detect via side channels, post-fabrication imaging techniques that can identify such Trojans have been proposed [152]. Lastly, our implemented metrics do not capture the threat of via-only additive Trojans. A via-only attack shorts two vertically-adjacent wires for the purpose of leaking information. I feel the possibility of such pernicious attacks in the future highlights the importance of ICAS's extensibility.

[3] Via-only attacks are outside the scope of our metrics as they are currently implemented (§3.8.6).

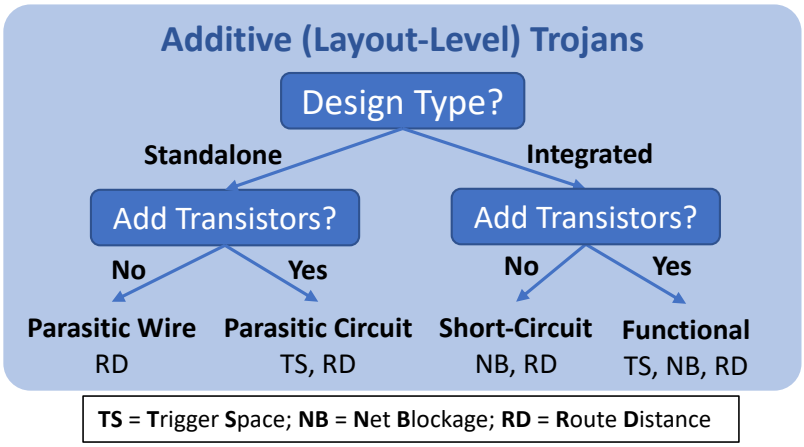

Figure 3.10: **ICAS Coverage of Trojans.** I assume that, at the very least, layout-level additive Trojans require adding rogue wires to the layout[3]. Whether the Trojan design is *integrated* (requires connecting to a host circuit) or *standalone*, or requires *additional transistors*, the difficulty of inserting it into a victim IC layout can be captured by our three metrics: 1) Trigger Space (TS), 2) Net Blockage (NB), and 3) Route Distance (RD).

3.8.7 Justification for Metrics

As a first step in estimating risk, I chose to implement three metrics that capture our decade worth of experience in implementing hardware Trojans: *net blockage*, *trigger space*, and *route distance*. These metrics capture the challenges I faced when inserting various types of additive Trojans into circuit layouts, i.e., *Trojan logic placement*, *victim/Trojan integration*, *intra-Trojan routing*. To facilitate mapping our metrics to specific Trojans I provide a taxonomy in Figure 3.10. To summarize the taxonomy: if a Trojan needs to attach to a victim wire (i.e., an integrated Trojan), our Net Blockage metric provides coverage; if the Trojan requires transistors to implement logic, our Trigger Space metric provides coverage; and if the Trojan needs to be near the victim wire (for capacitive coupling in the case of a standalone Trojan or to meet timing requirements in the case of a integrated Trojan), our Route Distance metric provides coverage. Additionally, as our evaluation with existing Trojans and real IC layouts shows, our metrics are both Trojan- and IC-layout-sensitive. Lastly, the metrics are hardware design agnostic. While I do not suggest that the implemented metrics are all-encompassing, our results suggest that these metrics are a

viable first step towards estimating a circuit's susceptibility to additive hardware Trojans.

3.9 Related Work

Fabrication-time attacks and defenses have been extensively researched. Attacks have ranged in both size and triggering-complexity [17, 90, 101, 142, 196]. Defenses against these attacks include: side-channel analysis [3, 13, 75, 117], imaging [2, 209], on-chip sensors [48, 97], and preventive measures [9, 10, 40, 195]. The most pertinent attacks and defenses are highlighted below.

3.9.1 Untrusted-foundry Attacks

The first foundry-level attack was conceived by Lin *et al.* [101]. This hardware Trojan was comprised of approximately 100 additional logic gates and designed to covertly leak the keys of an AES cryptographic accelerator using spread spectrum communication to modulate information over a power side channel. While the authors only demonstrated this attack on an FPGA, they are the first to mention the possibility of this type of Trojan circuit being implanted at an untrusted foundry.

The A2 attack [196] is the most recent fabrication-time attack. A2's analog triggering mechanism is stealthy, controllable, *and* small. It prevents the Trojan from being exposed during post-fabrication testing, or unintentionally through common usage. The attack requires only two additional standard cells and evades every known detection mechanism to date. ICAS quantifies the defensive coverage to these and other fabrication-time attacks.

3.9.2 Untrusted-foundry Defenses

Most untrusted foundry defenses rely on post-fabrication *detection* schemes [2, 3, 13, 48, 75, 97, 117, 209]. ICAS aims to guide innovation in *preventive* defenses against fabrication-time attacks, for which few mechanisms currently exist [9, 10, 40, 195]. I highlight some of these preventive measures and how ICAS could measure their effectiveness.

While preventive security-by-design was first explored at the behavioral (RTL) level by of Jin *et al.* [74], Xiao *et al.* were the first to demonstrate security-by-design at the layout-level with their BISA (Built-In Self-Authentication) scheme [195]. The *undirected* BISA approach attempts to eliminate *all* unused space on the device layer placement grid, and create routing congestion, by filling the device layer with interconnected tamper-resistant fill cells. Alternatively, recognizing the impracticality of filling 100% of the empty placement sites in complex circuit designs, Ba *et al.* take a *directed* approach to filling empty placement cites [9, 10]. Specifically, they only fill empty placement sites in close proximity to security-critical nets.

3.10 Conclusion

ICAS is an extensible framework that I use to expose and quantify gaps in existing defenses to the threat posed by an untrusted foundry. ICAS has two high-level components: *Nemo*, a tool that bridges the semantic gap across IC design processes by tracking security-critical signals across all stages of hardware development and *GDSII-Score*, a tool that estimates the difficulty a foundry-level attacker faces in attacking security-critical logic. Experiments with over 60 IC layouts across three open-source hardware cores and four foundry-level hardware Trojans reveal that all current defenses leave the IC design vulnerable to attack—and some are totally ineffective. These results show the value of a tool like ICAS that can help designers identify and address defensive gaps.

From a high level, ICAS is momentus in that it makes security a first-class concern during IC layout (in addition to power, area, and performance): ICAS allows IC designers to measure the security implications of tool settings and design decisions. ICAS fits well with existing IC design tools and flows, allowing them to consider security. ICAS is a critical measurement tool that enables the systematic development of future physical-level defenses against the threat of an untrusted foundry.

3.11 Citation

Work from this chapter was partially completed while interning at MIT Lincoln Laboratory, and is co-authored by Kang G. Shin, Kevin B. Bush, and Matthew Hicks. This work appeared in the 2020 IEEE Symposium on Security and Privacy, and can be cited as [169].

CHAPTER IV

T-TER

4.1 Introduction

Integrated circuits (ICs) are the foundation of computing systems. Security vulnerabilities in silicon are devastating as they subvert even formally verified software. For almost 50 years, the transistors within ICs have continued to shrink, enhancing performance while reducing power and area usage. However, these advances that push the laws of physics come with a financial cost: the price to build a 3 *nm* fabrication facility capable of producing ICs at a commercial scale is estimated to be $15–20B [94]. Even when entities can afford to make such an investment, they must continually run the IC fabrication line (approximately 40,000 wafers/month) as many fabrication processes cannot be readily stopped and restarted.

This extreme cost forces most semi-conductor companies, and even nation states, to become "fabless", i.e., they outsource fabrication. Today, only 3 companies in the world (Intel, Samsung, and TSMC) have capabilities to fabricate ICs at the 10/7 *nm* process nodes [95]. This presents a security threat: fabless semiconductor companies and nation states must trust these three manufacturers (and their partners) not to alter their designs at any point throughout the fabrication process (i.e., implant a hardware Trojan).

The most stealthy and controllable hardware Trojans involve inserting additional[1] cir-

[1] Additive hardware Trojans are a class of Trojan designs that require additional hardware to be added to a

cuit components designed to maliciously subvert the functionality of the chip (i.e., an additive hardware Trojan). Specifically, the A2 Trojan [196] utilizes only two additional cells—one analog capacitor and one digital logic gate— to provide a hardware foothold [85] within a microprocessor IC for an attacker to gain unauthorized supervisor privileges with user-mode code.

There are now only two ways of defending against hardware Trojans implanted at fabrication-time: post-fabrication *detection* [3, 48, 75, 97, 131, 209] and pre-fabrication *prevention* [9, 195]. The former tries to detect the presence of Trojan components after the chip has been fabricated, while the latter attempts to alter the IC's physical layout, at design time, in a way that makes foundry-side alterations challenging to an attacker.

Detection is more commonly studied than prevention and consists primarily of two techniques [162]: 1) side-channel analysis and 2) functional testing. Side-channel analysis attempts to detect noticeable deviations in power usage, electromagnetic (EM) emanations, performance (timing), etc. [3, 75, 117, 131]. It often requires a "golden" reference chip to be effective, and can only detect the side-channel signature deviations greater than those caused by process variation (i.e., the hardware Trojan must have a large physical footprint). Alternatively, functional testing attempts to inadvertently trigger the Trojan by activating as many logic paths through the circuit as possible. Functional testing does not require any "golden" reference chip, but it requires the Trojan's trigger to be activated by the IC's common mode operation, as exhaustive testing of even a moderately complex integrated circuit is infeasible.

Albeit less studied, *prevention* is another defense against fabrication-time hardware Trojans. To prevent such attacks, I advocate that the *placement and routing of security critical circuit elements should be a first-class part of an IC's back-end design*, on the level of performance, power, and cost. To the best of my knowledge, only three preventive

circuit design. I are unaware of any documented stealthy and controllable subtractive or substitution Trojans. Dopant-level Trojans [17, 90, 142] are the closest to such; however, they have limited controllability and are detectable [152].

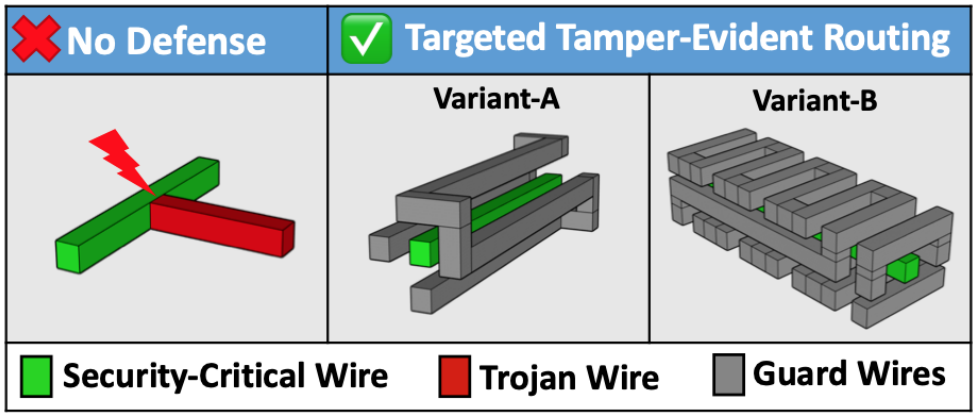

Figure 4.1: T-TER is a *preventive* layout-level defense against fabrication-time Trojans. T-TER deploys tamper-evident guard wires around security-critical wires in a circuit layout—in a pattern similar to variant A or B—to prevent attackers from attaching Trojan wires to them.

fabrication-time defenses have been explored [9, 10, 195]. All of them are *placement*-centric, attempting to increase the device layer (core) density by filling empty spaces with with tamper-evident logic gates, thus making it challenging for an attacker to find open space in the design to insert their Trojan components (cells/gates). However, there are several problems with placement-centric defenses. As Ba *et al.* [10] point out, the BISA cell approach [195] is infeasible as it requires 100% placement density. Contrast this with the 60-80% density of current IC layouts that ensures routability. If 100% density were feasible, every IC design would be manufactured that way to save cost. Alternatively, Ba *et al.* [9, 10] suggest targeted filling: only filling placement sites that are located closest to "security-critical" logic. While prioritizing security-critical logic is a significant improvement, focusing on the device layer only impedes attacks due to inflated timing requirements, it does not prevent them, as §4.6.2.2 shows.

Unfortunately, no single technique is effective in detecting, and/or preventing the insertion of the stealthiest known additive hardware Trojan, the A2 Trojan [196], which requires only two additional cells. To fill this gap, I propose *Targeted Tamper-Evident Routing (T-TER)*, a *routing-centric* defense that *prevents* foundry-side attackers from routing Trojan wires to, or directly adjacent to, security-critical wires. I define T-TER as any routing

method that protects security-critical wires from fabrication-time alterations. Specifically, I leverage concepts from the signal-integrity domain [60, 61] and apply them to a security domain (addressing several technical challenges along the way): I route "guard wires" around security-critical wires that make it infeasible for an attacker to tap any such wire without detection (i.e., tamper-evident), something characteristic of additive Trojans [169] (Fig. 4.1). Extending signal-integrity domain techniques to the security domain entails two technical challenges:

1. *completely* shielding all surfaces of critical wires,

2. and be tamper-evident.

Contrary to placement-centric defenses, which focus on preventing attack *implementation*, T-TER focuses on preventing attack *integration*, and thus, does *not* require filling *all* the empty space in an IC design to be effective.

I make the following contributions:

- Targeted Tamper-Evident Routing (T-TER): a routing-centric, preventative, defense against stealthy IC fabrication-time attacks. T-TER places *tamper-evident* guard wires alongside security-critical wires, making fabrication-time modifications to such wires infeasible and/or detectable post-fabrication.
- Characterization of possible guard wire bypass attacks.
- Attack-driven design of *designed-in guard wires*. Designed-in guard wires are added during the place-and-route phase of the IC design process for the sole purpose of defending security-critical wires. They have minimal routing constraints and can guard all surfaces of designer-targeted wires.
- Automated routing toolchain for deploying guard wires within an IC layout that integrates with commercial and open-source VLSI CAD tools.
- Evaluation of the effectiveness of T-TER compared to previous defenses against both digital and analog A2 Trojans embedded in a System-on-Chip intended to be a sur-

rogate for DoD systems of interest [108], using a recently published fabrication-time threat assessment tool [169]. The results indicate T-TER is more effective than existing placement-centric defenses [9, 10, 195], and is capable of thwarting even the stealthiest additive hardware Trojans, including A2 [196].[2]

4.2 Background

4.2.1 Fabrication-Time Attack Steps

As described in §3.2.2, implanting a hardware Trojan into an IC layout at fabrication-time requires three steps [169]:

1. Trojan Placement,
2. Victim/Trojan Integration, and
3. Intra-Trojan Routing.

Trojan Placement is the process of finding empty space on the IC's device layer (Fig. 4.2) to add additional circuit components, e.g., logic gates, to construct the Trojan trigger and payload. *Victim/Trojan Integration* requires attaching a rogue Trojan wire, or routing it directly adjacent, to an unblocked surface on a security-critical wire(s). Lastly, *Intra-Trojan Routing* involves routing the Trojan circuit components to the Victim/Trojan integration point—the unblocked security-critical wire segment.

4.2.2 Layout-Level Defenses.

Prior work attempts to thwart fabrication-time attacks by increasing the difficulty of *Trojan Placement*: filling empty space on the IC's device layer with temper-evident functional logic gates [9, 10, 195]. As shown in [169], this approach is only effective for Trojans

[2]It is important to note that routing-centric and placement-centric defenses are compatible (belt and suspenders). A designer would first apply T-TER, then fill open placement sites in a targeted manner.

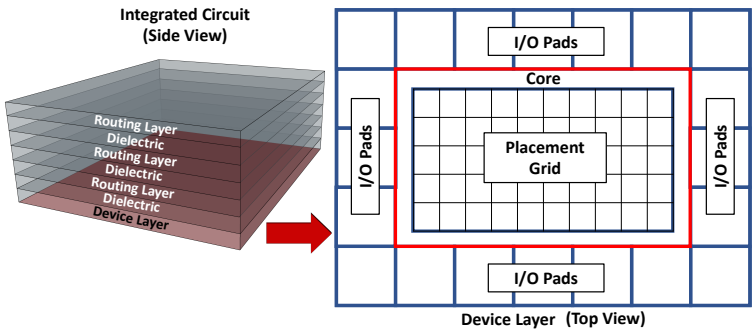

Figure 4.2: **Three Dimensional IC Layout.** Typical 3D physical IC layout designed during the place-and-route IC design phase (Fig. 2.1). On the bottom is a device layer, and stacked above are several routing layers.

with large footprints, as filling all placement sites is infeasible [10], and even targeting fill around security-critical logic [9] leaves the IC layout vulnerable to Trojans with small footprints [196]. Orthogonally, *T-TER targets Victim/Trojan Integration by directing protection, at the routing level, around wires Trojans want to attach to.*

4.2.3 Time-Domain Reflectometry (TDR)

Time-domain reflectometry (TDR) is an electrical analysis technique used to measure physical characteristics about a transmission line (i.e., a wire) such as length, number and distance between impedance discontinuities (e.g., bends), propagation delay, dielectric constant, etc. [56, 64]. Foundries already use TDR to perform root cause analysis on chips that fail post-fabrication testing—often during bring-up of a new process node. TDR works by characterizing a wire within a circuit by injecting a single rising pulse down that wire and analyzing its reflection(s).

4.2.4 IC Interconnect Models

There are two ways to model IC wires (interconnect): lumped and transmission-line models [12]. Lumped interconnect models approximate interconnects using networks of resistors and capacitors. Transmission-line models approximate interconnects as transmission lines with a characteristic impedance and propagation delay.

The choice of interconnect model is a function of maximum frequency component to wire length [153]. A common rule of thumb for IC interconnects is: *a wire is considered a transmission line if its length is greater than ≈10% of the wavelength of the maximum frequency component it transmits [153].* In digital electronics, it is common to think of signals in terms of rise and fall times, rather than maximum frequency component. Thus, one can modify the prior rule of thumb to: *a wire is considered as a transmission line if the transmitted signal rise time, T_{rise}, is less than twice the wire's propagation delay, T_{pd} [153].* Eq. (4.1) captures this rule of thumb.

$$\text{Model} = \begin{cases} \text{Transmission Line}, & T_{rise} < 2T_{pd} \\ \text{Lumped RC}, & \text{otherwise} \end{cases} \tag{4.1}$$

Choosing the right model is vital to understanding operational limitations and ensuring signal integrity within an IC layout. For example, an interconnect that carries a high-speed signal transitions will observe signal reflections from impedance discontinuities that are destructive to the signal integrity of the overall system. Modeling such interconnects using a lumped RC model can hide these destructive effects, while a transmission-line model would not.

4.2.5 TDR for IC Fault Analysis

By Eq. (4.1), the faster the rising edge of TDR's incident pulse, the finer-grain of propagation delay changes are detectable. TDR was first developed as a fault-analysis technique for long transmission lines, such as telephone or optical communication lines [129, 146]. As commercial TDR systems became more advanced, TDR became a standard IC packaging fault analysis tool [36, 119, 144]. Researchers have now demonstrated terahertz- level TDR systems capable of locating faults in IC interconnects to nanometer-scale accuracies [30, 115, 160, 164]. With such fine-grain resolution, **TDR is an ideal tamper-analysis technique for ensuring the integrity of the guard wires** used in T-TER (§4.6.4).

4.3 Threat Model

Again, in this chapter, I adopt the threat model for fabrication-time attacks—§2.3.1 and Figure 2.3A. While there are many types of hardware Trojans [135] (§2.2), in this chapter, I again focus on additive Trojans, rather than subtractive or substitution Trojans. Additive Trojans require implanting additional circuit components and wiring into the IC design. I focus on additive Trojans as there are no documented stealthy and controllable examples of subtractive or substitution Trojans that I are aware of. The closest example of such Trojans are dopant-level Trojans [17, 90, 142], all of which have limited controllability and are detectable with optical microscopy [152].

Previous work shows that to successfully implement an *additive* hardware Trojan, the adversary must complete the three steps—*Trojan Placement*, *Victim/Trojan Integration*, and *Intra-Trojan Routing* [169]—without being exposed. Namely, they must 1) find empty space on the device layer to insert the Trojan's components (logic gates/cells), 2) locate an unblocked segment on a security-critical wire to attach the Trojan to, and 3) route the Trojan components to that unblocked wire segment. They are restricted from modifying the dimensions of the chip and/or violating manufacturing design rules that would risk their exposure. They are allowed to move components and/or existing wiring around, but are constrained by available resources (e.g., time) and correctness from making mass perturbations to the layout. As process technologies scale, manufacturing design rules become increasingly complex [147]. Thus, rearranging components and/or existing wiring comes at a substantial cost. The time to complete any layout modifications, and verify such modifications have not violated design correctness, cannot disrupt the fabrication turn-around time expected by their customers.[3] Additionally, the attacker avoids any modifications that are detectable using existing test-case or side-channel based defenses. While it would be trivial for an attacker with *infinite* time and resources to reverse-engineer the physical layout into HDL, add a Trojan, and re-run the design through the entire IC design process

[3]Typically, fabrication turn-around times are ≈3 months [93, 173].

(Fig. 2.1) thus generating an entirely new layout, such an attack will be infeasible within the hard time limits of fabrication contracts, thus outside the scope of our threat model.

4.4 Targeted Tamper-Evident Routing (T-TER)

T-TER aims to make the second step of Trojan insertion—*Victim/Trojan Integration* (§4.2.1)—intractable by shielding the surfaces of targeted wires (interconnects) with tamper-evident guard wires (§4.2.2), creating an additional obstacle for adversaries to overcome. Similar to prior work [9, 10, 102, 169], T-TER is made practical by leveraging the observation that, for most hardware designs, only a subset of the IC is security-critical [59, 74, 102, 167, 205, 207], or the target of a hardware Trojan. In designing T-TER, I pose three questions:

1. *Which wires in the design are security-critical (should be guarded)?*
2. *How can an attacker bypass T-TER guard wires?*
3. *How do I design guard wires that are tamper-evident with respect to bypass attacks?*

4.4.1 Identifying Security-Critical Nets to Guard

While identifying security-critical *features* in a design is an orthogonal problem—and an ongoing area of research [59, 74, 102, 167, 205, 207]— identifying the *nets (wires)* that comprise said features is the first step in deploying T-TER. Currently, there exist two techniques for identifying security-critical nets: 1) *manual* [59, 74, 102] or 2) *semi-autonomous* [205, 207]. In *manual* identification, a human expert analyzes the design's specification, and the corresponding HDL, and flags nets that implement features critical to the security of software or other hardware that interface to the design [59, 74, 102]. Alternatively, in *semi-autonomous* identification, a set of security-critical nets for a specific design are first manually identified [59, 74], or mined from a list of published errata [205], and either: 1) used to train a classifier that identifies similar nets in other designs [205], 2) expanded using information flow [167] or fan-in analyses [169], or 3) translated to an

entirely different design [207]. In this chapter, I adopt the most common approach in this area of semi-autonomous identification [9, 169].

4.4.2 Guard Wire Bypass Attacks

With T-TER deployed, attackers must *bypass* guard wires—by exposing the surface of a security-critical wire(s)—to complete *Victim/Trojan Integration*, i.e., connect a rogue Trojan wire to a security-critical wire(s) (§4.2.1). Given a set of interconnected guard wires (Fig. 4.1), there are three ways an attacker can bypass them, color-coded by attacker difficulty (Fig. 4.3): A) delete, B) move, or C) jog attacks. In a *deletion* attack (Fig. 4.3A), entire guard wire(s) are removed from the layout. While this attack is easy to implement, it is also easy to defend. A post-fabrication continuity check of a connected set of guard wires will detect a deletion attack. In a *move* attack (Fig. 4.3B), all interconnected guard wires are left intact, but translated to another location on the chip. Move attacks are the most difficult to implement: an attacker must find a contiguous group of unused routing tracks to translate each set of guard wires too. Even then, a post-fabrication cross-talk analysis between security-critical and guard wires would expose this attack [60, 131]. Lastly, in a *jog* attack, guard wires are *lengthened* to make room for a rogue Trojan wire to connect to a security-critical wire using a via. Jog attacks strike a compromise in terms of implementation difficulty, and are the stealthiest of all bypass attacks. They are easier to implement than move attacks, and are undetectable with post-fabrication continuity tests or cross-talk analyses. *The only artifacts of a jog attack are: 1) a change in the number of bends in the guard wire, i.e. number of impedance discontinuities, and/or 2) an increase in the guard wire's length.* However, nanometer scale TDR [115, 160] detects these changes (§4.6.4).

4.4.3 Tamper-Evident Guard Wires

While techniques for detecting all three bypass attacks exist, each of them requires the ability to measure physical characteristics (e.g., continuity, cross-talk, and length) about a

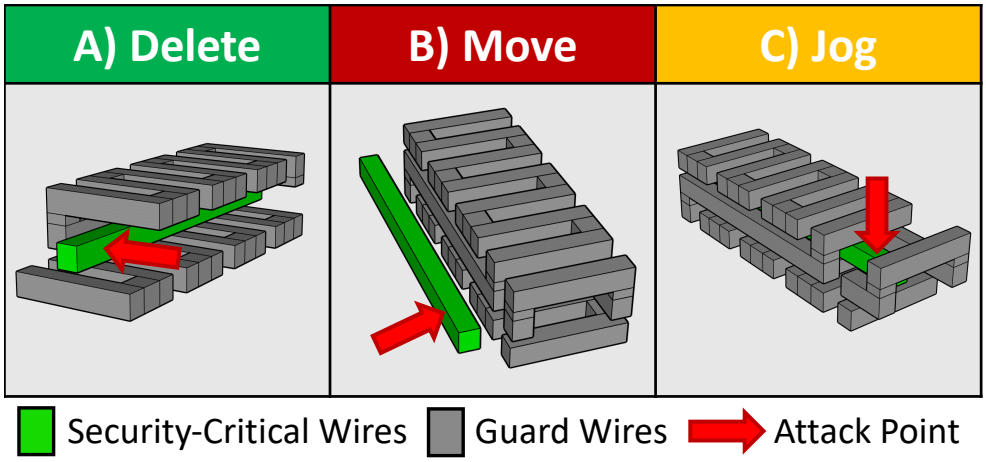

Figure 4.3: There are three ways an attacker could bypass T-TER guard wires to connect a Trojan wire to a security-critical wire, color-coded by attacker difficulty: A) *delete* guard wire(s), B) *move* an intact set of guard wires, or C) *jog* guard wires out of the way. I study the *jog* attack to assess defensive sensitivity, as it strikes a balance in attacker difficulty, and is the most difficult to detect.

guard wire post-fabrication. *How do I design guard wires whose physical characteristics are tamper-evident post-fabrication?* Based on these considerations, I take a straw-man approach in designing guard wires capable of preventing even the stealthiest of attacks.

4.4.3.1 Naïve Approach: Re-purpose Existing Wires

One idea for constructing guard wires is to re-purpose existing non-security-critical wires, inherent to the host IC design, as guard wires. Such an approach creates hyper-local routing densities nearby security-critical wires, thus limiting or eliminating the locations where an attacker can attach rogue Trojan wires. By re-purposing pre-existing wires as guard wires, the guard wires incur no hardware overhead. Unfortunately, there are additional routing constraints (e.g., toggle frequency, length, layer, location, timing sensitive, and spacing) that limit the pool of candidate guard wires. Even when such constraints are met, the guard wires are only tamper-evident with respect to deletion and move attacks. For an existing wire to also be tamper-evident with respect to the more stealthy jog and bypass attacks, it must be timing-critical (i.e., if it is made longer, then it will cause timing

violations that manifest as run-time errors). As Fig. 4.5 shows, deployment using existing guard wires is challenging. Namely, the lack of suitable wires in many designs makes it infeasible to block all surfaces of all security-critical wires.

4.4.3.2 Designed-in Guard Wires

To fill the gaps of existing wires, I propose designed-in guard wires. Designed-in guard wires are **not** inherent to the host IC design. Rather, they are added to the design during the place-and-route IC design phase (Fig. 2.1). Since they do not implement any circuit functionality, they have fewer routing constraints. As I show in Fig. 4.5, completely blocking the accessible surface area of all security-critical wires is trivial. While designed-in guard wires incur hardware overhead, i.e., additional wires, they completely block an attacker from attaching a Trojan wire at fabrication time (Victim/Trojan Integration, §4.2.1), as shown in Fig. 4.6. Additionally, designed-in guard wires are tamper-evident with respect to ***all*** bypass attacks, when coupled with post-fabrication analysis techniques like continuity checking, cross-talk analysis, and time-domain reflectometry (§4.2.3 and §4.6.4), respectively.

There are several designed-in guard wire architectures that may be deployed, listed in order of increasing difficulty of deployment: 1) fully-disjoint, 2) partially-connected, and 3) fully-connected. Fully-disjoint designed-in guard wires are not connected between sides, i.e., the guard wires on each side of a security-critical wire are never connected to one another. Partially-connected guard wires allow for a single guard wire to be utilized on multiple sides. For example, a security-critical wire could be guarded on the north, east, and west sides by a single guard wire that wraps around the security-critical wire. Lastly, fully-connected guard wires are formed when a single guard wire is routed around all sides of all security-critical wires, as shown in Fig. 4.1.

To detect tampering of designed-in guard wires post-fabrication, their analog characteristics of must be observable. This can be implemented either on-chip, e.g., with internal

sensors [82] or ring oscillators [208], or off-chip, e.g., with two I/O pins and a one-time programmable fabric [102]. If fully-joint or partially-connected designed-in guard-wires are deployed, the one-time programmable fabric could be randomly programmed to route both ends of a single (fully-disjoint) or single-set (partially-connected) of guard wire(s) to the two pins. If fully-connected designed-in guard wires are deployed, the one-time programmable fabric is not needed, as both ends of the guard wires set can be routed to the two pins.

4.5 Implementation

I develop an automated toolchain for deploying T-TER in modern IC designs. My toolchain integrates with existing IC design flows (Fig. 2.1) that utilize commercial VLSI CAD tools. Specifically, I implement the T-TER toolchain around the Cadence Innovus Implementation System [25], a commercial place-and-route (PaR) CAD tool. The toolchain is invoked by modifying a place-and-route TCL script,[4] as shown in Fig. 4.4.

4.5.1 Place-&-Route Process

The PaR design phase (Fig. 2.1) is typically automated by a CAD tool, programmatically driven by TCL script(s). There are several steps to PaR that are performed in the following order: 1) floor-planning, 2) placement, 3) clock tree synthesis, 4) routing, and 5) filling. *To ensure that all guard-wires are routed optimally, I modify the order of these PaR steps*. Specifically, after floor-planning (1), I use my automated toolchain to place identified components *and* route identified wires and their guard wires. My toolchain then permanently fixes the locations of these components and wires to prevent the PaR CAD tool from modifying their positions and/or shapes throughout the remainder of the PaR process. Lastly, I utilize the PaR CAD tool to place all other components (2), synthesize the clock

[4]Tool Command Language (TCL) scripts are the standard programmatic interface to commercial VLSI CAD tools. IC designers often develop a set of scripts for driving the CAD tools that automate most of the IC design process (Fig. 2.1).

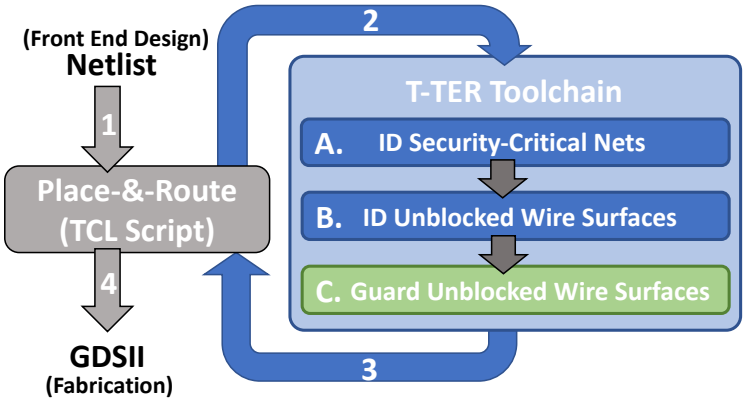

Figure 4.4: T-TER is an automated toolchain consisting of three phases. My toolchain first identifies which wires are security-critical, determines potential (unblocked) attachment points, and routes guard wires to block all attachment points. Identified components & wires are placed & routed *before* phase (A) of my toolchain is invoked. Before continuing with the traditional PaR flow, the protected nets and their guard wires are locked in-place to ensure they are untouched throughout the remainder of the layout process.

tree (3), route remaining wires(4) and fill the design with filler (capacitor) cells.

4.5.2 Automated Toolchain

The T-TER toolchain automates the insertion of either existing or designed-in guard wires around wires in need of protection. The toolchain consists of three main phases (Fig. 4.4). The first phase (A) identifies security-critical nets. The second phase (B) identifies the unblocked surfaces of all of these nets within a GDSII-encoded layout. The last phase (C) guards the nets and their influencer nets by routing guard wires nearby. I provide additional implementation details on all three stages of the T-TER toolchain below.

4.5.2.1 Identifying Nets.

The first phase of T-TER requires identifying nets in the design to guard, i.e., nets that are security-critical. Phase A of my toolchain (Fig. 4.4A) utilizes a semi-autonomous approach to identifying such nets (§4.4.1). Specifically, my toolchain assumes the designer has *manually* flagged a set of *root* security-critical nets in the behavioral-level HDL by ap-

pending a unique prefix—*secure_*—to each signal (net) name. During PaR, my toolchain performs a data-flow analysis of the circuit netlist to locate the direct fan-in—to a configurable depth—of each root net. Since the netlist is often modified by PaR CAD tools to meet various design constraints (e.g., power, performance, and area), I disable the optimization of all root nets during PaR. Given the interconnected nature of nets within an IC design, an adversary may elect to target a net that influences a root net, rather than the root net itself. My toolchain addresses this indirection, using an autonomous approach that widens the set of *targeted* nets to the root nets and those that influence root nets (to a designer configurable degree). The remainder of my tool flow focuses on protecting this set of targeted nets.

My fan-in analysis tool is a custom-backend to the Icarus Verilog (IVL) front-end Verilog compiler [192], and is implemented in C++. It performs a breadth-first search over the circuit-level data-flow graph generated by IVL. I release my fan-in analysis tool under an open-source license.

4.5.2.2 Identifying Unblocked Wire Surfaces.

The second phase of T-TER is identifying the unblocked surfaces of targeted nets in a physical IC layout, i.e., potential locations of Trojan wire attachment. To do so, I implement, and open-source, a Python tool that analyzes the GDSII layout file containing only the placed-and-routed targeted components and wires. My tool implements a 3-D scanning window approach to search the 3-D boundary surrounding each targeted wire, and compute the areas on each wire's surfaces that are not blocked by other wires or circuit components. While it is traditional for designers to only route wires on defined *routing tracks*, i.e., on a pre-defined routing grid, it may be possible for an attacker to route Trojan wires off this grid, so long as they maintain the minimum spacing requirements dictated by the manufacturing design rules. Thus, my tool takes a conservative approach when scanning for unblocked wire surfaces, only scanning the 3-D boundary surrounding each targeted wire

that extends up to the minimum-spacing requirements defined for the given, and adjacent (top/bottom), routing layers. If and only if another component or wire overlaps a region of the 3-D boundary surrounding a targeted wire, that surface region will be considered blocked. The output of this stage of my toolchain is a list of coordinates within the 3-D place-and-route grid that must be filled with guard wires during the next phase in the T-TER toolchain.

4.5.2.3 Guard Unblocked Wire Surfaces.

The last stage of the T-TER toolchain (Fig. 4.4) is a custom guard wire routing tool, also implemented in Python. It takes as input exact locations of targeted wires and their unblocked sides (output from Phase B, §4.5.2.2) and generates a TCL script that integrates with the Cadence Innovus Digital Implementation platform [25] to automatically route the guard wires. This TCL script is executed immediately after the targeted wires have been routed, but before placing the remaining components. Depending on the guard wires being deployed, existing or designed-in, different guard wire TCL scripts are generated (described below).[5] Note, in either case, my toolchain routes guard wires that are compliant with all manufacturing design rules.

There are numerous ways *existing guard wires* can be implemented. Since commercial PaR CAD tools do not offer an interface to enable fine-grain constraints between two unrelated signal wires, I develop an indirect method for implementing existing guard wires. I implement existing guard wires by constraining placement and routing resources nearby targeted wires. First, I identify all circuit components (i.e., logic gates) connected to all targeted wires, i.e., targeted components. Next, I draw a bounding box around these components and extend this boundary vertically by $Y\%$ of the overall box height, and horizontally by $X\%$ of the overall box width. Then, I set placement and routing density screens in

[5] While existing guard wires fail to defend against all types of guard wire attacks (§4.4.3.1), I implement a tool to deploy them in order to empirically show they are also inferior to designed-in guard wires in terms of surface-are coverage (Figs. 4.5 & 4.6), and thus should not be used in a security context.

the portion of the IC layout that lies *outside* the bounding box. These constraints limit the placement and routing resources outside the bounding box, thus forcing more components and wiring within the bounding box. With increased routing density nearby targeted wires, they are less accessible by Trojan payload delivery wires. The values of X, Y, and density screen configuration settings are optimized to maximize the net blockage metric computed by the GDS2Score metric.

Designed-in guard wires are more straightforward to implement. The automated guard wire deployment toolchain locates all unblocked surfaces (north, south, east, west, top, and bottom) of all targeted wires and routes guard wires in these regions. After all guard wire segments are routed, they are connected according to the architecture chosen (§4.4.3.2).

4.6 Evaluation

I evaluate T-TER in three areas. First, I explore the effectiveness of T-TER at closing the fabrication-time attack surface of three security-critical features within an open-source System-on-Chip (SoC), with regard to the stealthiest additive Trojan known: the A2 Trojan [196]. I compare the capabilities of T-TER with existing state-of-the-art layout-level defenses [9, 10, 195]. Next, I demonstrate the practicality of T-TER, analyzing its power, performance, and area overheads. Finally, I perform a threat assessment, demonstrating how guard wires are tamper-evident.

4.6.1 Experimental Setup

4.6.1.1 Surrogate SoC

I utilize the open-source Common Evaluation Platform (CEP) SoC design [109] for my evaluation. The CEP platform is designed as a surrogate SoC system for testing a variety of DoD-oriented IC technologies. It contains a general-purpose processor core, five cryptographic cores, four digital signal processing cores, and a GPS core. I focus on three

cores from in the SoC: the *processor* core, the *DFT* core, and the *AES* core. The OR1200 processor[6] is a 5-stage pipelined CPU that implements a 32-bit OR1K instruction set and Wishbone bus interface [121], and is the same design used in previous fabrication-time attack studies [169, 196]. It supports Linux via BusyBox [178]. The AES core supports 128-bit key sizes. The DFT accelerator implements a Discrete Fourier Transform algorithm, a common component of radar and other sensing systems.

I target a 45 *nm* Silicon-On-Insulator (SOI) process technology with 10 available routing layers. I synthesize my design with Cadence Genus (v16.23), and placed-and-route it using Cadence Innovus (v17.1). All layout variations of my SoC target a 100 MHz clock frequency and a core density of 60–80%. All CAD tools are run on a server with 2.5 GHz Intel Xeon E5-2640 CPU and 64GB of memory, running Red Hat Enterprise Linux (v6.9).

4.6.1.2 A2 Trojan

The goal of T-TER is to protect security-critical features within SoCs from the stealthiest additive Trojan currently known, the A2 Trojan [196]. The A2 Trojan is stealthy, i.e., evades current *prevention* and *detection* defenses, due to its small size and complex triggering mechanism. When implemented within my surrogate SoC, in a 45 *nm* process, the analog variant of the A2 Trojan [196] requires only two additional cells that occupy 20 placements sites, while the entirely digital variant of the same attack requires 91 additional cells that occupy 1,444 placement sites. The analog A2 attack is *not* timing critical: the Trojan components may be placed anywhere on the placement grid, at any distance from the Victim/Trojan integration point. Conversely, the digital A2 attack *is* timing-critical: the length of the interconnect between the Trojan components and the Victim/Trojan integration point must be within three standard deviations from the mean net length in the

[6] I use the OR1200 version of the CEP rather RISC-V version since the OR1200 is the processor used in the A2 Trojan [196]. I are not aware of similar Trojans available in the RISC-V. I expect similar results for the RISC-V version of the CEP since both processors are RISC-based, in-order, scalar, pipelined, capable of running Linux, and operate at similar clock frequencies. Thus, from an IC layout perspective, they have similar features (e.g., wire lengths) and will have similar hardware overheads.

Table 4.1: A2 Trojans used in T-TER effectiveness assessment.

Trojan	# Std Cells	# Placement Sites	Timing Critical?
A2 Analog [196]	2	20	✗
A2 Digital [196]	91	1444	✓

overall SoC (this is an entirely worst-case estimate borrowed from [169]). I summarize the placement and routing resource requirements for the two variants of the A2 Trojan in Table 4.1.

4.6.1.3 Exemplar Nets of Interest

For this evaluation, I need to protect nets that my example Trojan might want to use as integration points. Leveraging existing hardware Trojan payloads, I select three reference integration targets within my SoC design to protect with T-TER:

1. processor supervisor bit (supv),
2. DFT computation ready interrupt (next_out),
3. cryptographic key bits (key [0:127]).

The most popular hardware Trojans leverage the supervisor (supv) net as part of privilege escalation attacks [58, 85, 196]. Alternatively, hardware Trojans can also hide specific computations or state transitions, e.g., a Trojan that disables the DFT computation-ready interrupt signal (or *next_out* signal) that informs the CPU when a DFT computation is ready. Lastly, another popular hardware Trojan seeks to leak cryptographic key bits via side channels [101]. The A2 trigger can be attached to any of the nets that carry these signals to mount an attack, so I protect the interconnects that comprise these nets.

The initial stage (Fig. 4.4A) of my automated T-TER toolchain assumes the designer has manually annotated the root nets they have chosen to target with T-TER (§4.5.2.1). Thus, I manually annotate the above net (signal) definitions with the prefix *secure_* within my SoC design's RTL. I then synthesize and place-and-route my design prior to generating a final, optimized, netlist for which my toolchain computes the fan-in to each manually

annotated net—to a depth of two layers of logic gates—thereby expanding the final set of all targeted nets (i.e., those guarded by T-TER). Fig. 4.7 (far right) shows the number of interconnect wires that comprise each set of nets that implement the aforementioned features within my surrogate SoC.

4.6.2 Effectiveness

I first evaluate the effectiveness of T-TER in thwarting the insertion of hardware Trojans at fabrication time. I compare the degree of protection provided by T-TER with that provided by deploying the current state-of-the-art *preventive* defense suggested by Ba *et al.* [9, 10]. This placement-based defense involves filling as many empty placement sites as possible (they show filling 95% of all placement sites is the max feasible), prioritizing empty sites nearest security-critical nets. I use my automated toolchain (§4.5.2) to deploy both types of guard wires (existing and designed-in). I assume the best case scenario for Ba *et al.'s* placement defense [9, 10] by filling 95% of the device layer with inverter cells—the smallest cells in my 45 *nm* cell library, for fine grain filling.

I use the ICAS framework [169] to quantify the effectiveness of each defense. ICAS analyzes the physical layout of an IC (encoded in a GSDII file), and computes security metrics detailing the IC layout's fabrication-time attack surface. Namely, it computes three metrics: 1) *trigger space*, 2) *net blockage*, and 3) *route distance*. The ***trigger space*** metric characterizes the open space on the device layer (empty placement sites) available for an attacker to add their Trojan components. The ***net blockage*** metric computes the percentage of surface area of identified nets that are blocked by other circuit components or wiring. Lastly, the ***route distance*** metric computes the minimal distance between unblocked identified nets and unused placement sites that an adversary would have to route a rogue Trojan wire to "connect" the hardware Trojan to the host IC. The trigger space metric quantifies the difficulty of performing *Trojan Placement*, the net blockage quantifies the difficulty of performing *Trojan/Victim Integration*, and the route distance metric quan-

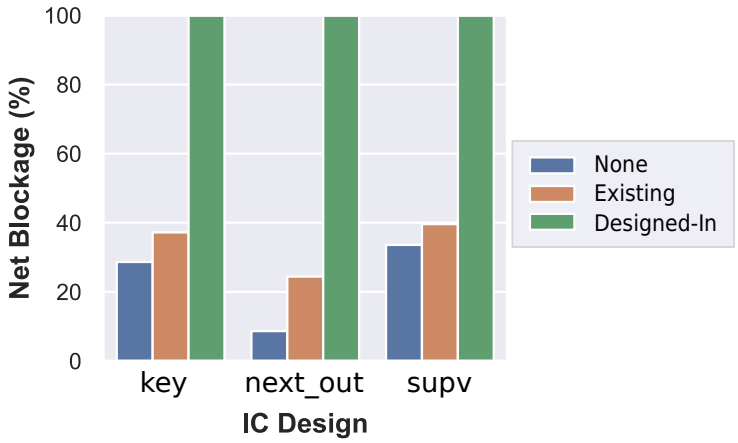

Figure 4.5: Plot of the *net blockage* [169] computed across three different sets of targeted nets within my SoC layout, with and without guard wires.

tifies the difficulty of performing *Intra-Trojan Routing* (§4.2.1). Of the three ICAS metrics, the *net blockage* metric is most adept to quantifying the deployability of each guard wire type (existing and designed-in), i.e., how effective each guard wire type is at shielding all targeted nets. Alternatively, the *route distance* metric is the adept at comparing T-TER with Ba *et al.*'s placement defense, as it is essentially a combination of the *trigger space* metric—an entirely placement-focused metric—and the *net-blockage* metric—an entirely routing-focused metric. Therefore, I utilize these two ICAS metrics in the following evaluation.

4.6.2.1 Net Blockage Results

Both existing and designed-in guard wires attempt to block targeted nets to prevent attackers from attaching rogue wires to them, thus minimizing/eliminating possible *Victim/Trojan Integration* points (§4.2.1). I use the *net blockage* metric to compute the surface-area-coverage differences between existing and designed-in guard wires. Fig. 4.5 compares the net blockage computed across three total IC layouts of the same SoC design, including: three guard wires variations—without guard wires, with existing guard wires, and with designed-in guard wires—across three different sets of targeted nets. All net-blockage re-

sults are with respect to each set of targeted nets in the SoC.

Across all three sets of targeted nets, designed-in guard wire provide more protection than existing guard wires, as expected. Specifically, for all nets, designed-in guard wires achieve 100% net blockage. This means that there is no place on any targeted net within the SoC where an attacker can attach a rogue wire. Existing guard wires are unable to achieve 100% coverage due mainly to having to meet their own routing constraints which prevents my tool from locating enough nets to block all surfaces of all targeted nets, making them ineffective at thwarting attacks.

4.6.2.2 Route Distance Results

Since T-TER only limits the routing resources needed to insert a Trojan at fabrication time, it is vital to understand how T-TER reduces the overall fabrication-time attack surface, i.e., both Trojan routing *and* placement resources. I use the *route distance* metric to locate all possible combinations of unused placement sites and unblocked targeted nets—i.e., all possible Trojan attack configurations [169]. I use the *route distance* metric to illustrate the attack surface across each core within my SoC where that contains the root net of interest. I analyze the *route distance* metric with respect to each containing core, as it is common practice for IC layout engineers to lay out each core separately, before integrating them, plus this increases the clarity of presentation.

Fig. 4.6 shows the *route distance* metric as computed across all three containing cores, with and without layout-level defenses including: 1) T-TER (both existing and designed-in guard wires) and 2) defensive placement. Each heatmap is intended to be analyzed column-wise, where each column is a histogram of the distances between unblocked targeted nets and trigger-spaces[7] within a size range. Namely, each heatmap illustrates the fabrication-time attack surface of each IC layout. If a circuit has no attack configurations, i.e., all targeted nets are blocked or there are no trigger-spaces, the route distance heatmap

[7]Trigger spaces are contiguous groups of placement sites that are empty, or contain (removable) capacitive fill cells [169]

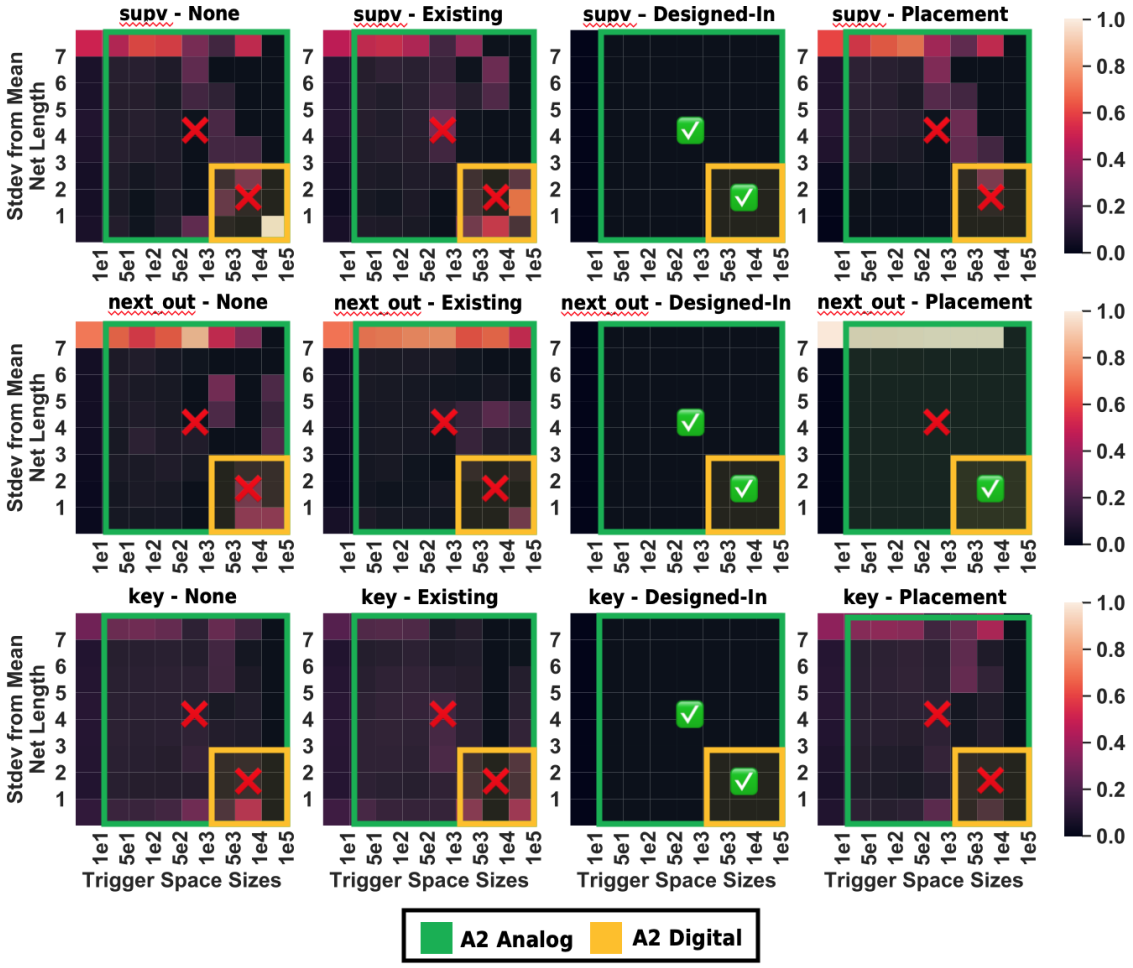

Figure 4.6: Plot of the ICAS *route distance* metric [169] computed across four different layouts of each core within my surrogate SoC, with and without guard wires and Ba *et al.*'s defensive placement [9, 10]. Each heatmap illustrates the percentage of (targeted net, trigger-space) pairs (possible Trojan layout implementations) of varying distances apart. The heatmaps are intended to be analyzed by column, as each column encodes a histogram of possible attack configurations with trigger-spaces of a given size range (X-axis). Route distances (Y-axis) are displayed in terms of standard deviations from mean net length in each respective design. Heatmaps that are completely dark indicate no possible attack configurations exist, i.e., no placement/routing resources to insert any Trojan. Overlaid on each heatmap are rectangles indicating regions on the heatmap a given A2 Trojan (Tab. 4.1) may exploit, and markers (checks and x-marks) indicating if a non-zero number of specific Trojan layout implementations are possible.

is completely dark (column ratios of 0). If it is impossible to eradicate all attack configurations, the most secure layout for such a circuit would have maximum distances between unblocked targeted net and trigger-spaces, i.e., a heatmap with the top row the lightest color (top row ratios of 1). This is because larger distances increase the signal delay for the hardware Trojan; increasing the challenge of the attacker to meet timing constraints for their attack. Overlaid on each heatmap are rectangles indicating the region of the attack surface that is exploitable by the color-coded Trojan, and check- or x-marks indicating whether any possible attack configurations exist for that attack. A check-mark indicates there are zero possible Trojan layouts (success)), where an x-mark indicates the opposite (vulnerable).

Designed-in guard wires outperform existing guard wires and placement-centric defenses. For all three example attack payloads, designed-in guard wires were able to close the fabrication-time attack-surface by *completely* blocking all targeted nets (Fig. 4.5). Therefore, even the stealthiest A2 Trojan [196] cannot be utilized to attack the features-of-interest within my SoC.

4.6.3 Practicality

T-TER is effective, but is it practical? I evaluate the cost of deploying T-TER across three exemplar security-critical features within my SoC that have been subject to attack. Specifically, I analyze the power, route density, and performance (timing) overheads incurred by deploying both existing and designed-in guard wires from §4.6.2. Note, while T-TER guard-wires can be deployed on any routing layer, I chose to prioritize routing security-critical nets on metal layers *three* and *four* (out of 10 total layers) to measure overheads in the worst case, i.e., guard wires routed on layers 2–5. Measurements are taken with respect to each feature's containing core, similar to the route distance measurement. While it is common to analyze power, performance, and area, of an IC design, I instead analyze power, performance and *route density*. Area measurements refer to the device-layer area, i.e., width and length, since the height (number of routing layers) is fixed for a given

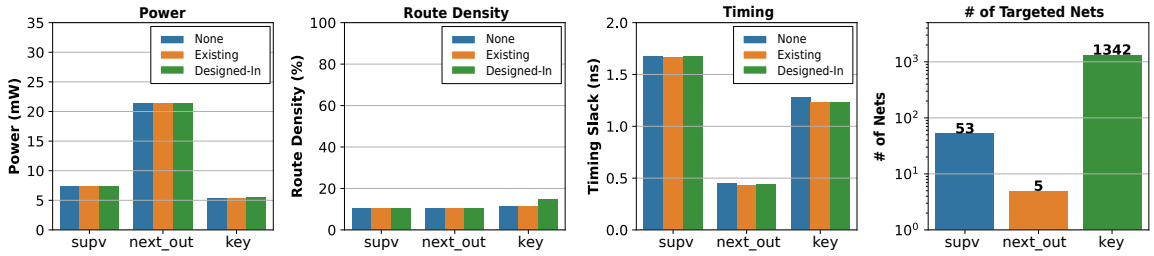

Figure 4.7: T-TER hardware overheads. The far right plot shows the number of wire (route) segments that implement the labeled security-critical feature (set of nets) in my surrogate SoC.

process technology. Since T-TER does not require additional logic gates, I do not increase the width and height (area) of the core area, rather T-TER alters the total wire length in the design. Thus, measuring routing density overhead is more meaningful. I use the built-in features of Cadence tools to compute these overheads.

Fig. 4.7 shows my results. Power and timing overheads were both less than 1%. In some cases, the timing was better for the guard wire designs. This is expected as T-TER does not require any additional logic gates, nor lengthen existing wires. Rather, the guard wires increase routing constraints that can push the PaR CAD tool to produce more optimal routing solutions. The route density overhead was less than 1% for all existing guard wires, and similar for designed-in guard wires when the number of targeted nets to guard is small, namely the *supv* and *next_out* nets. Intuitively, the more guard wires inserted, the higher the routing density increase. Keeping route density low is important to ensure automated CAD tools can route each design. However, even though all layouts targeted a placement density (density of logic gates on the device layer) of 60–80%, route density was relatively low even with guard wires. This was due to the characteristics of the designs and process technology (i.e., back-end-of-line metal stack option).

It is worth noting that in addition to low power, performance, and area overheads, deploying T-TER guard wires has minimal impact on the run-time of layout CAD tools. Without DR, the tools lay out each SoC core in less than 10 minutes, and with DR they lay out each core in less than 11 minutes. Tool run-time overheads are more impacted by the

magnitude of features requiring protection than on circuit complexity.

4.6.4 Threat Analysis of Bypass Attacks

Lastly, I provide a threat analysis of T-TER. Recall, of the three ways an attacker can bypass T-TER guard wires to carry out a fabrication-time attack (Fig. 4.3 and §4.4.2), the *jog* attack is the stealthiest. An attacker mounts a jog attack by *jogging*, or moving, a portion of a guard wire to a nearby routing track, in order to make room for a rogue Trojan wire to attach to a targeted net (Fig. 4.3C). In such an attack, the guard wire is *lengthened*, or *bends* are added/removed. To evaluate the detectability of such an attack, I ask three questions:

1. *What is the smallest jog attack, i.e., the minimum alteration in a guard wire's length and/or number of bends?*
2. *Is the smallest jog attack masked by process variation?*
3. *Can modern TDR detect the smallest jog attacks?*

4.6.4.1 Smallest Jog Attack

The minimum jog attack is to jog a top (or bottom) guard wire to an adjacent routing track, and attach to the targeted net from above (or below) with a via, as illustrated in Fig. 4.3C. This edit either increases the length of the guard wire, or adds/removes bends—impedance discontinuities—in the guard wire to keep its overall length unchanged. This edit is minimal because the minimal metal pitch (MMP), or (horizontal) distance between the centers of adjacent routing tracks on the *same routing layer*, is much smaller than the (vertical) distance between overlapping routing tracks on *adjacent routing layers*. Specifically, the smallest jog attack would either: 1) increase a guard wire's length by: $L_{attack} = 2 * MMP_r$, where MMP_r is the MMP on layer r, as defined in the design rules of a given process technology, or 2) add/remove bend(s) in the guard wire that are at least a

Table 4.2: Minimum guard wire jog attack (Fig. 4.3C) edit–distances for each routing layer in the IBM 45 *nm* SOI process technology.

Routing Layer	**Min Wire Spacing** (*um*)	**Min Metal Pitch** (*um*)	**Min Attack Edit** (*um*)	**TDR Detectable?**
1	0.07	0.14	0.28	✓
2	0.07	0.14	0.28	✓
3	0.07	0.14	0.28	✓
4	0.09	0.19	0.38	✓
5	0.09	0.19	0.38	✓
6	0.14	0.28	0.56	✓
7	0.14	0.28	0.56	✓
8	0.80	1.60	3.20	✓
9	0.80	1.60	3.20	✓
10	2.00	4.00	8.00	✓

distance of L_{attack} apart from existing bends. *In either case, a feature resolution—of overall length or length between bends—of L_{attack} is required to detect the smallest jog attack.* Table 4.2 summarizes the *minimal-attack-edits* (L_{attack} distances), to a guard wire's features an attacker must make to bypass T-TER, according to the 45 *nm* process technology I target in this study.

4.6.4.2 Process Variation vs. Smallest Jog Attack

Assume for a moment that I can measure the of overall length, or length between bends, of a guard wire to infinite accuracy. Even then, detecting the smallest jog attack requires the minimal attack edit distance, L_{attack}, be discernable from deviations between simulated and fabricated guard wire lengths due to process variation. Fortunately, L_{attack} **is larger than the worst-case manufacturing process variation** in a guard wire's length. Namely, with L_{design} as the designed length of the guard wire, and L_{wc_error}, as the worst-case manufacturing error in the actual guard wire's length (+ or −):

$$L_{design} - L_{wc\_error} + L_{attack} > L_{design} + L_{wc\_error} \tag{4.2}$$

For a guard wire on routing layer r, the *worst-case* manufacturing error, L_{wc_error}, can

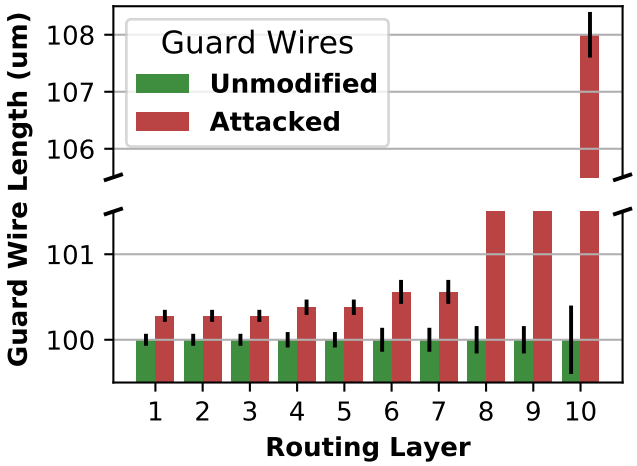

Figure 4.8: Worst-case manufacturing process variation (error bars) effect on unmodified and minimal jog attacks on 100-micron guard-wires.

be deduced from the manufacturing design rules as:

$$L_{wc\_error} = 2 * \frac{min\_spacing_r}{2} = min\_spacing_r \tag{4.3}$$

where $min\_spacing_r$ is the minimum required spacing surrounding a wire routed on metal layer, r.

I illustrate this in Fig. 4.8, where I plot the minimum length differences between unmodified (un-attacked) and minimally-jogged (attacked) guard wires, overlaid with error bars indicating the worst-case range of variation in a guard wires fabricated length caused by process variation. Even in the worst case, across all routing layers, unmodified vs attacked guard wires are discernible.

4.6.4.3 Attack Detection with TDR

When IC interconnects are injected with a pulsed waveform with a rise time less than twice the propagation delay of the interconnect, they behave like transmission lines (Eq. (4.1)). Hence, time-domain reflectometry (TDR) can be used to measure several characteristics of

designed-in guard wires to ensure they have not been tampered with (§4.2.3). Specifically, the *lengths* of each guard wire, or *lengths between bends* on each guard wire, are computed by measuring the reflection time(s) of a single incident rising pulse applied to the guard wires under test. Once measured, the lengths can be compared with that predicted by a 3D electromagnetic field solver [100] to detect if they have been altered. While modeling *all* interconnects within a large complex IC using a field solver is computationally impractical, it is *practical* to analyze only a small subset of interconnects, e.g., the guard wires and surrounding circuit structures [115].

Prior work demonstrates terahertz TDR systems [30, 115, 160, 164] capable of measuring the propagation delay of an interconnect to a resolution of ± 2.6 femptoseconds (fs). Such systems utilize laser-driven optoelectronic measurement techniques to achieve such high resolutions. According to the ideal transmission line model [153], the propagation delay, T_{pd}, is a function of the dielectric constant, D_k, speed of light, C, and **length of the transmission line (guard wire)**, L_{gw}, as shown in Eq. (4.4).

$$T_{pd} = L_{gw} * \frac{\sqrt{D_k}}{C} \tag{4.4}$$

TDR is the ideal tamper detection tool as process variation has no impact on its *accuracy*. Knowing the dielectric constant, D_k, of the insulating material surrounding the guard wires—the inter-layer dielectric (ILD)—is *all* that is required to compute their lengths, or the lengths between their bends (Eq. (4.4)). Since, the dielectric constant of the ILD is **not** dependent on its geometric properties, it is well controlled [21].

Using the TDR propagation delay model described in Eq. (4.4), and the previously studied resolution of optoelectrical terahertz TDR [30, 115, 160, 164], I simulate the detection of the smallest jog attacks on guard wires across every routing layer in my target 45 nm process. Namely, I simulate the difference in reflection times observed for single pulse TDR waveforms applied to (unmodified) guard wires that are 100 microns long, compared to the reflection time observed from similar guard wires that have been lengthened by the

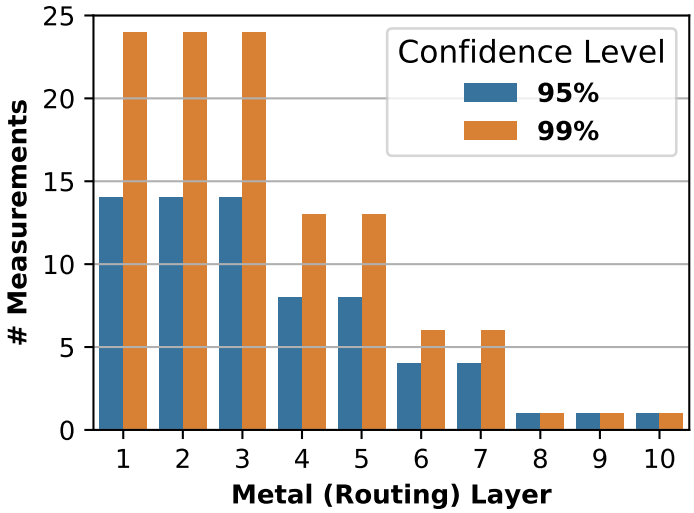

Figure 4.9: Number of TDR measurements required to detect the smallest jog attacks (Table 4.2) with 95% and 99% confidence, per layer.

minimal attack edit distances, L_{attack}, across each routing layer (Table 4.2). I assume a dielectric constant of 3.9, the nominal dielectric constant of silicon dioxide [86]. Taking into account a (Gaussian) standard error (across reflection time measurements) of $\pm 2.6\ fs$, as reported by [115], I compute the minimum number of TDR measurements required to discriminate an unmodified guard wire from an attacked guard wire with confidence levels of 95% and 99%. I plot these results in Figure 4.9. My results demonstrate that existing terahertz TDR systems are capable of detecting the smallest jog attacks across all routing layers (Table 4.2) in my target $45\,nm$ process, requiring at most 14 and 24 TDR measurements to achieve confidence levels of 95% and 99%, respectively.

4.7 Discussion

T-TER aims to prevent fabrication-time Trojan attacks that target specific security-critical features in an IC design. Experiments on real circuit layouts of a SoC containing show that T-TER is effective, deployable, and tamper-evident. Discussed below are the limitations, scalability, signal integrity impact, flexibility, and extensibility of T-TER.

4.7.1 Limitations

T-TER is a mitigation strategy for hardware designs where only a subset of the design is security-critical [59, 167]. As my evaluation results show, the deployability and performance overhead of T-TER is low when the overall security-critical wire length is low. If *every* wire in a design is security-critical, then T-TER is not a good defensive strategy; in fact, the motive for outsourcing fabrication in such scenarios is tenuous. If fabrication must be outsourced, I recommend alternative mitigation strategies such as those proposed in [9, 10, 68, 102, 195]. The tradeoff is that these strategies have limited deployability, and a large, fixed, performance overhead that make them impractical for designs that require only a subset of security-critical functionality be protected.

4.7.2 Scalability

There are two notions of scalability to address. The first is scalability with regard to *routability*. Routing guard wires alongside security-critical wires can impact the routability of a layout, if the 1) ***percentage of overall wire length*** to guard, and 2) ***route-density without guard wires*** are both large. By placing and routing security-critical components and wires first, before any other portions of the circuit (§4.5.1), I are able to minimize security-critical wire length. This makes security-critical wire length scale with the total length of security-critical wires, as opposed to the size of the overall design. As I see when going from OR1200 and RISC-V class processor to modern x86-64 processors, the proportion of security-critical functionality (hence wires) decreases as relatively more transistors are spent on performance. Moreover, by deploying T-TER within advanced process nodes—which is the motivating threat model—route density is minimized since these nodes provide multiple metallization options[8] with 10 (or more) routing layers. To demonstrate this empirically, I highlight the AES core (Fig. 4.7–Route Density), where I guard over 1000

[8]The *metallization option* defines the total number (and physical characteristics) of available routing (metal) layers defined within an IC's process technology.

nets with little impact on power or performance. In fact, the reason I select the AES as a benchmark—even though it is arguably entirely security-critical—is because ***its key-bit nets exhibit a unique quality that stress tests T-TER***. Specifically, they are global, highly-connected routes that are orders-of-magnitude longer than any other nets in the layout.

The second notion of scalability is with regard to the detection of bypass attacks. Although Moore's law is near its limit, transistors continue to shrink. Only three companies in the world are capable of manufacturing 7–10 *nm* transistors [95]. It is, therefore, vital for T-TER to scale with process technology. With respect to deletion attacks (Fig. 4.3A), T-TER scales with process technology advances as measuring interconnect continuity does not differ across process technologies. With respect to move attacks (Fig. 4.3B), T-TER scales with process technology advances as cross-talk is amplified when interconnects are smaller and more densely packed. Lastly, with respect to jog attacks, T-TER also scales, as TDR capabilities directly scale with microelectronic feature sizes, i.e., faster transistors translates to faster TDR rise times.

4.7.3 Signal Integrity Impact

Routing long wires parallel to targeted nets increases coupling capacitance, thus creating cross-talk between the guard wires and the targeted nets they protect. However, designed-in guard wires are not actively driven during normal chip operation, and can be permanently grounded (using a one-time programmable fabric) after TDR analysis. Thus, cross-talk is not an issue—in fact, designed-in guard wires decrease cross-talk by acting as shields between targeted nets and the rest of the circuit.

4.7.4 Defense-in-Depth

While T-TER alone can thwart even the stealthiest fabrication-time attacks, its low deployment costs also enable defense-in-depth. Layering T-TER with other preventive measures, such as Ba *et al.*'s defensive placement [9, 10], provides an additional layer of

protection.

4.7.5 Extensibility of CAD Tools

My T-TER deployment framework (§4.5) is built on top of a commercial IC CAD tool [25] and an open-source VLSI analysis tool [169]. Extending T-TER to work across other commercial IC layout CAD tools involves incorporating support for each vendor's CAD tool APIs. I foresee T-TER deployed as an integrated component of commercial VLSI CAD tools as they focus more on IC security.

4.8 Related Work

Fabrication-time attacks and defenses have been extensively studied. Attacks have spanned the trade-space of footprint size, stealth, and controllability. Specifically, some attacks have demonstrated stealth and controllability, at the cost of large footprints [17, 85, 101], while others have demonstrated small (or non-existent) footprints, at the cost of controllability and stealth [90, 142]. The most formidable attack—the A2 attack [196]—has demonstrated all three: small footprint, stealth, and controllability. I highlight a few notable attacks and defenses below.

On the defensive side, there are two main strategies: detective or preventive. Most prior work has focused on detective strategies, while few works have focused on preventive strategies. Detective strategies involve side-channel analysis [3, 13, 75, 117], imaging [2, 209], and on-chip sensors [48, 62, 97]. Until T-TER, preventive measures have been placement-focused [9, 10, 195].

Fabrication-time Attacks. The first fabrication-time insertion of a hardware Trojan was developed by Lin *et al.* [101] who proposed a Trojan designed to leak information over a deliberately created side channel. Specifically, they designed and implemented a hardware Trojan, with a footprint of approximately 100 logic gates, to create an artificial power side channel for leaking cryptographic keys. Albeit unique at the time, today such a

large footprint makes the attack detectable via side channel defenses [3, 13, 48].

The most lethal fabrication-time attack is the A2 Trojan, developed by Yang *et al.* [196]. The A2 Trojan utilizes analog components to build a counter-based trigger circuit with a footprint of less than the size of one flip-flop. Its complex triggering mechanism makes it stealthy, i.e., unlikely to accidentally deploy during post-fabrication functional testing or under normal chip operation, yet is controllable from user-level software. Its unique design makes it the only Trojan to evade all detection schemes, except T-TER.

Fabrication-time Defenses. The first side-channel detection scheme was proposed by Agrawal *et al.* [3]. They used power, temperature, and electromagnetic (EM) side-channel measurements to record a fingerprint of a "golden" IC during normal, and compared this fingerprint to one acquired from an untrusted IC. Similarly, Jin *et al.* [75] create a timing-based fingerprint obtained by measuring the output delays resulting from applying various input combinations to a given IC. While side-channel detection schemes are effective against hardware Trojans with large footprints, they fail at detecting Trojans like A2 [196], whose side-channel signatures are well below operational noise margins.

Of all fabrication-time Trojan defenses, R2D2 [62] is the only one that claims to detect the A2 Trojan. R2D2 works by using on-chip sensors to monitor the toggling frequency of a select few security-critical signals within the design. If the toggling rate of any security-critical signals exceed a pre-determined threshold, then an alarm signal is activated to indicate an A2 Trojan may have been triggered. The crux of this approach is that, unlike T-TER guard wires, the hardware used to construct the toggle frequency monitors *is not tamper-evident*. There is no way to tell if a foundry-side attacker disabled the R2D2 hardware while inserting her Trojan.

4.9 Conclusion

T-TER is a routing-centric preventive defense against additive fabrication -time Trojans that target security-critical hardware features. It makes routing Trojan wires to, or directly

adjacent to, attacker-targeted wires in a victim IC intractable by surrounding their surfaces with tamper-evident guard wires. I propose the use of designed-in guard wires in conjunction with post-fabrication terahertz time-domain reflectometry (TDR) analysis to detect *all* bypass attacks I contrive (*deletion*, *move*, and *jog* attacks). I develop an automated toolchain for deploying T-TER guard wire. Lastly, I evaluate the effectiveness, deployability, and tamper-evidence of T-TER at securing multiple security-critical features within an SoC that have been subject to attack by existing hardware Trojans. My results show that T-TER thwarts the insertion of even the stealthiest known additive hardware Trojan—the A2 Trojan—with power, timing, and area overheads of $\approx 1\%$.

4.10 Citation

Work from this chapter was partially completed while interning at MIT Lincoln Laboratory, and is co-authored by Kang G. Shin, Kevin B. Bush, and Matthew Hicks. This work can be cited as [170].

CHAPTER V

Bomberman

5.1 Introduction

As microelectronic hardware continues to scale, so too have design complexities. To design an IC of modern complexity targeting a 7 *nm* process requires 500 engineering years [53, 92]. Because it is impractical to take 500 years to create a chip, semiconductor companies reduce time-to-market by adding engineers: increasing both the size of their design teams and their reliance on 3rd-party IP. Namely, they purchase pre-designed blocks for inclusion in their designs, such as CPU cores and cryptographic accelerators (e.g., AES). This year, analysts estimate that a typical System-on-Chip (SoC) will contain over 90 IP blocks [19]. From a security perspective, this reduces trust in the final chip: with an increased number of (both in-house and external) designers molding the design, there is increased opportunity for an attacker to insert a hardware Trojan.

Hardware Trojans inserted during design time are both *permanent* and *powerful*. Unlike software, hardware cannot be patched in a general-purpose manner; repercussions of hardware flaws echo throughout the chip's lifetime. As hardware vulnerabilities like Meltdown [103], Spectre [88], and Foreshadow [174] show, replacement is the only comprehensive mitigation, which is both costly and reputationally damaging. Moreover, vulnerabilities in hardware cripple otherwise secure software that runs on top [85]. Thus, it is vital that hardware designers verify their designs are Trojan-free.

Prior work attempts to detect hardware Trojans at both design and run time. At design time, researchers propose static (FANCI [184]) and dynamic (VeriTrust [202] and UCI [58]) analyses of the RTL design and gate-level netlists to search for rarely-used circuitry, i.e., potential Trojan circuitry. At run time, researchers: 1) employ hardware-implemented invariant monitors that dynamically verify design behavior matches specification [59, 182], and 2) scramble inputs and outputs between trusted and untrusted components [183] to make integration of a hardware Trojan into an existing design intractable. These attempts to develop general, "one-size-fits-all", approaches inevitably leave chips vulnerable to attack [150, 181, 203].

Verifying a hardware design is Trojan-free poses two technical challenges. First, hardware Trojan designs use the same digital circuit building blocks as non-malicious circuitry, making it difficult to differentiate Trojan circuitry from non-malicious circuitry. Second, it is infeasible to exhaustively verify, manually or automatically, even small hardware designs [125], let alone designs of moderate complexity. These challenges are the reason why **"one-size-fits-all" approaches are incomplete and akin to proving a design is bug-free**.

Instead of verifying a design is free of *all* Trojan classes, I advocate for a divide-and-conquer approach, breaking down the RTL Trojan design space and systematically ruling out each Trojan class. I begin this journey by eliminating the most pernicious RTL hardware Trojan threat: the TTT. As Waksman *et al.* state [182, 183], when compared with other stealthy design-time Trojans (i.e., data-based Trojans), TTTs provide "the biggest bang for the buck [to the attacker] ... [because] they can be implemented with very little logic, are not dependent on software or instruction sequences, and can run to completion unnoticed by users." Moreover, TTTs are a flexible Trojan design in terms of deployment scenarios. **An attacker looking to deploy a TTT does not require any *a priori* knowledge of how the victim circuit will be deployed at the system level, nor post-deployment (physical or remote) access to the victim circuit [182, 183]**. By eliminating the threat of TTTs, I mimic the attack-specific nature of system-level software defenses

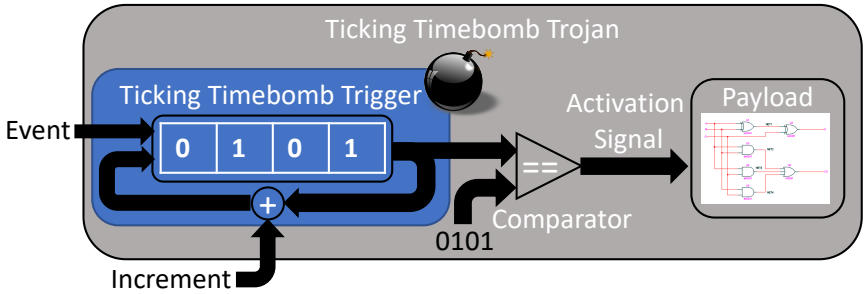

Figure 5.1: **Ticking Timebomb Trojan (TTT).** A TTT is a hardware Trojan that implements a ticking timebomb *trigger*. Ticking timebomb triggers monotonically move closer to activating as the system runs longer. In hardware, ticking timebomb triggers maintain a non-repeating sequence counter that increments upon receiving an event signal.

like Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) in hardware, i.e., I force RTL attackers to implement Trojan designs that require post-deployment attacker interaction. This is the hardware analog to defending against data injection attacks in software, forcing attackers to employ more complex data reuse attacks; a necessary part of a comprehensive, layered defense.

To ensure my defense is systematic and avoids implicit assumptions based on existing TTTs, I first define an abstract TTT based on its behavior. At the heart of any TTT is a trigger that tracks the progression of values that form some arbitrary sequence. The simplest concrete example is a down-counter that releases the attack payload when it reaches `zero`. Thus, I define TTTs as devices that track an arbitrary sequence of values constrained by only two properties:

- **the sequence never repeats a value,**
- **the sequence is incomplete.**

Fig. 5.1 shows the basic hardware components required to implement such a sequence counter in hardware. It has three building blocks: 1) **SSCs**, 2) an **increment value**, and 3) an **increment event**.

To understand the power my definition gives to attackers, I use it to enumerate the space of all possible TTT triggers. I define a total of six TTT variants, including *distributed* TTTs

that couple together SSCs scattered across the design to form a sequence counter and *non-uniform* TTTs that conceal their behavior by incrementing with inconsistent values, i.e., expressing what looks like a random sequence.

I leverage my definition of TTTs to locate SSCs in a design that behave like TTT triggers during functional verification. Specifically, I reduce the Trojan search space of the DUT by analyzing *only* the progression of values expressed by SSCs of potential TTT triggers. I design and implement an automated extension to existing functional verification toolchains, called **Bomberman**, for identifying the presence of TTTs in hardware designs. Bomberman computes a DFG from a design's HDL (either pre- or post- synthesis) to identify the set of all combinations of SSCs that could construct a TTT. Initially, Bomberman assumes all SSCs are *suspicious*. As Bomberman analyzes the results obtained from functional verification, it marks any SSCs that violate my definition as *benign*. Bomberman reports any remaining *suspicious* SSCs to designers, who use this information to create a new test case for verification, or manually inspect connected logic for malice.

I demonstrate the effectiveness of Bomberman by implanting all six TTT variants into four different open-source hardware designs: a RISC-V CPU [193], an OR1200 CPU [121], a UART [121] module, and an AES accelerator [136]. Even with verification simulations lasting less than one million cycles,[1] Bomberman detects the presence of *all* TTT variants across *all* circuit designs with a false positive rate of less than 1.2%.

This chapter makes the following contributions:

- An abstract definition and component-level breakdown of TTTs.
- Design of six TTT variants, including new variants that evade existing defenses.
- Design and implementation of an automated verification extension, Bomberman, that identifies TTTs implanted in RTL hardware designs.
- Evaluation of Bomberman's false positive rate and a comparative security analysis against a range of both TTT-focused and "one-size-fits-all" design-time hard-

[1] Typical verification simulations last $\approx$ millions of cycles [182].

ware Trojan defenses; Bomberman is the only approach capable of detecting all TTT variants, including state-of-the-art pseudo-random [191] and non-deterministic [69] TTTs.

- Algorithmic complexity analysis of Bomberman's *SSC Enumeration* and *SSC Classification* stages.
- Open-source release of Bomberman and TTTs [168].

5.2 Background

5.2.1 Design-Time Hardware Trojans

In this chapter I shift my focus to *design-time attacks* (§2.3.2), i.e., detecting hardware Trojans that are inserted at design-time [85, 101, 182, 183]. In Figure 5.2, I refine the hardware Trojan taxonomy previously presented in Figure 2.2 to make characterizations according to 1) where in the IC design process they are inserted, and 2) their *trigger* architectures [81, 162].

There are two main types of triggers: *always-on* and *initially dormant*. As their names suggest, *always-on* triggers indicate a triggerless Trojan that is always activated, and are thus trivial to detect during testing. Always-on triggers represent an extreme in a trigger design trade-space—not implementing a trigger reduces the overall Trojan footprint at the cost of sacrificing stealth. Alternatively, *initially dormant* triggers activate when a signal within the design, or an input to the design, changes as a function of normal, yet rare, operation, ideally influenced by an attacker. *initially dormant* triggers enable stealthy, controllable, and generalizable hardware Trojans. As prior work shows, it is most advantageous for attackers to be able to construct triggers that hide their Trojan payloads to evade detection during testing [58, 69, 183, 184, 202], so I focus on *initially dormant* triggers.

Initially dormant triggers consist of two sub-categories: *data-based* and *time-based* [69, 182, 183]. Data-based triggers, or *cheat codes*, wait to recognize a single data value (single-

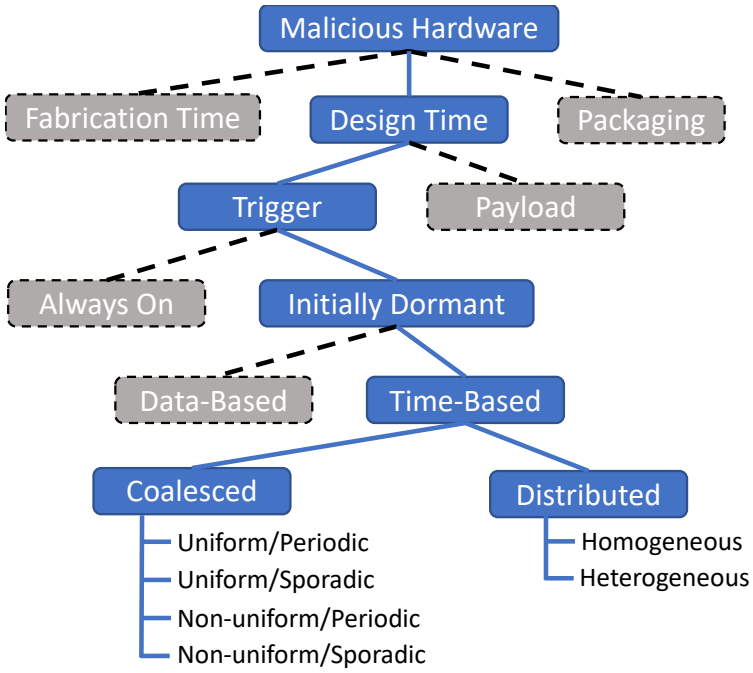

Figure 5.2: **Taxonomy of Hardware Trojans.** Hardware Trojans are malicious modifications to a hardware design that alter its functionality. I focus on time-based Trojans (TTTs) and categorize them by design and behavior.

shot) or a sequence of data values to activate. Alternatively, time-based triggers, or *ticking timebombs*, become increasingly more likely to activate the more time has passed since a system reset. While, ticking timebombs can implement an indirect and probabalistic notion of time (§5.4), a simple ticking timebomb trigger is a periodic up-counter, where every clock cycle the counter increments, as shown in Fig. 5.3A. In this work, I eliminate the threat of TTTs to force attackers to implement data-based Trojans that require post-deployment attacker interaction to trigger [182].

5.3 Threat Model

In this chapter, I focus on the *design-time attack* threat model (§2.3.2). Additionally I further constrict this threat model, focusing on identifying TTTs, as I define them (§5.4), and leave the identification of other Trojan types to existing heuristics-based [58, 184, 202], and future design-specific defenses. My defense can be deployed at any point throughout

the front-end design process—i.e., directly verifying 3rd party IP, after RTL design, or after synthesis—after which the design is trusted to be free of TTTs.

5.4 Ticking Timebomb Triggers

First, I define TTTs by their behavior. Based on this definition, I synthesize the fundamental components required to implement a TTT in hardware. Finally, using these fundamental components I enumerate six total TTT variants, including previously contrived TTTs that resemble contiguous time counters [182, 183], to more complex, distributed, non-uniform, and sporadic [69, 191] designs.

5.4.1 Definition

I define TTTs as the set of hardware Trojans that implement a time-based trigger that monotonically approaches activation as the victim circuit continuously operates without reset. ***More succinctly, I define a ticking timebomb trigger based on two properties of the values it exhibits while still dormant yet monotonically approaching activation***:

Property 1: The TTT does NOT repeat a value without a system reset.

Property 2: The TTT does NOT enumerate all possible values without activating.

Property 1 holds by definition, since, if a TTT trigger repeats a value in its sequence, it is no longer a ticking timebomb, but rather a data-based "cheat code" trigger [182, 183]. Property 2 holds by contradiction in that, if a TTT trigger enumerates all possible values *without* triggering, i.e., no malicious circuit behavior is observed, then the device is not malicious, and therefore not part of a TTT. Upon these two properties, I derive the fundamental hardware building blocks of a TTT.

Figs. 5.3A–D illustrate example ticking timebomb behaviors that are captured by my definition, in order of increasing complexity. The most naive example of a ticking timebomb trigger is a simple periodic up-counter. While effective, a clever attacker may choose

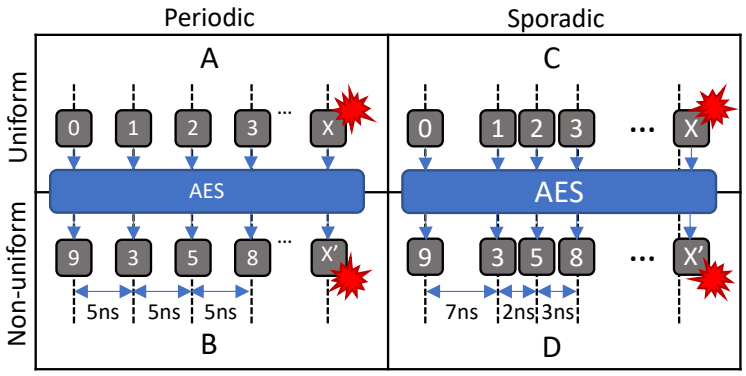

Figure 5.3: **Ticking Timebomb Trigger Behaviors.** There are four primitive ticking timebomb trigger counting behaviors, in order of increasing complexity, captured by my definition (Properties 1 & 2 in §5.4.1). **A)** The simplest counting behavior is both *periodic* and *uniform*. Alternatively, more sophisticated counting behaviors are achieved by: **B)** encrypting the count to make the sequence *non-uniform*, **C)** incrementing it *sporadically*, or **D)** both.

to hide the monotonically increasing behavior of a *periodic* up-counter by either 1) obscuring the relationship between successive counter values (e.g., AES counter mode sequence, Fig. 5.3B), or 2) *sporadically* incrementing the counter (e.g., a non-deterministic TTTs [69], Fig. 5.3). Even more sophisticated, the attacker may choose to do both (Fig. 5.3D).

5.4.2 TTT Components

From my definition, I derive the fundamental components required to implement a TTT in hardware. Fig. 5.1 depicts these components. For TTTs to exhibit the behaviors summarized in Fig. 5.3, they must implement the notion of an *abstract time counter*. TTT time counters require three components to be realized in hardware: 1) **State-Saving Components (SSCs)**, 2) increment **value**, and 3) increment **event**.

The *SSC* defines how the TTT saves and tracks the triggering state of the time counter. SSCs can be either *coalesced* or *distributed*. Coalesced SSCs are comprised of *one* N-bit register, while distributed SSCs are comprised of M, N-bit registers declared across the design. Distributed SSCs have the advantage of increasing stealth by combining a subset of one or multiple coalesced SSCs whose count behaviors *individually* violate the definition of

a TTT trigger (i.e., Properties 1 and 2), but when considered *together* comprise a valid TTT. Distributed SSCs can also reduce hardware overhead through reuse of existing registers.

The TTT *increment value* defines *how* the time counter is incremented upon an increment event. The increment value can be uniform or non-uniform. Uniform increments are hard-coded values in the design that do not change over time, e.g., incrementing by one at every increment event. Non-uniform increments change depending on device state and operation, e.g., incrementing by the least-significant four bits of the program counter at every increment event.

Lastly, the TTT *increment event* determines *when* the time counter's value is incremented. Increment events may be *periodic* or *sporadic*. For example, the rising edge of the clock is periodic, while the rising edge of an interrupt is sporadic.

5.4.3 TTT Variants

From the behavior of the fundamental TTT components, I extrapolate six TTT variants that represent the TTT design space as I define. I start by grouping TTTs according to their SSC construction. Depending on their sophistication level, the attacker may choose to implement a simplistic *coalesced* TTT, or construct a larger, more complex, *distributed* TTT. If the attacker chooses to implement a *coalesced* TTT, they have four variants to choose from, with respect to increment uniformity and periodicity. The most naive attacker may choose to implement a coalesced TTT with uniform increment values and periodic increment events. To make the coalesced TTT more difficult to identify, the attacker may choose to implement non-uniform increment values and/or sporadic increment events.

To increase stealth, an attacker may choose to combine two or more coalesced TTTs, that alone violate the definition of being a TTT trigger, but combined construct a valid *distributed* TTT. An attacker has two design choices for distributed TTTs. Seeking to maximize stealth, the attacker may choose to combine several copies of the *same* coalesced TTT with non-uniform increment values and sporadic increment events, thus implementing

a *homogeneous* distributed TTT. Alternatively, the attacker may seek integration flexibility, and choose to combine various coalesced TTTs to implement a *heterogeneous* distributed TTT. For homogeneous distributed TTTs, an attacker has the same four design choices as in coalesced TTTs. However, for heterogeneous distributed TTTs, the design space is much larger. Specifically, the number of sub-categories of heterogeneous distributed TTTs can be computed using the binomial expansion, $\binom{n}{k}$, with n, the number of coalesced sub-triggers, and k, the number of unique sub-trigger types. I summarize all six TTT variants and their behaviors in Figs. 5.2 and 5.3, respectively, and provide example implementations in Verilog below.

In the following Verilog examples of TTT triggers, I use a three letter naming convention to describe their building blocks: SSC type (C or D), increment value (U or N), and increment event (P or S). For example, a CNS TTT indicates a *Coalesced* (C) SSC, with a *Non-uniform* (N) increment value, and a *Sporadic* (S) increment event. For TTTs comprised of *distributed* SSCs I use the "D-<type>" naming convention to indicate the type: homogeneous or heterogeneous. This list is not comprehensive, but rather a representative sampling of the TTT design space. Note, all examples assume a processor victim circuit, with a *pageFault* flag, *overflow* flag, and a 32-bit *program counter* (PC) register.

```
1  // 1. CUP = Coalesced SSC, Uniform increment, Periodic event
2  reg [31:0] ssc;
3  always @posedge(clock) begin
4      if ( reset )
5          ssc <= 0;
6      else
7          ssc <= ssc + 1;
8  end
9  assign doAttack = (ssc == 32'hDEAD_BEEF);
```

```
1  // 2. CUS = Coalesced SSC, Uniform increment, Sporadic event
2  reg [31:0] ssc;
3  always @posedge(pageFault) begin
4      if ( reset )
5          ssc <= 0;
6      else
7          ssc <= ssc + 1;
8  end
9  assign doAttack = (ssc == 32'hDEAD_BEEF);
```

```
// 3. CNP = Coalesced SSC, Non-uniform increment, Periodic event
reg [31:0] ssc;
always @posedge(clock) begin
    if ( reset )
        ssc <= 1;
    else
        ssc <= ssc << PC[3:2];
end
assign doAttack = (ssc == 32'hDEAD_BEEF);
```

```
// 4. CNS = Coalesced SSC, Non-uniform increment, Sporadic event
reg [31:0] ssc;
always @posedge(pageFault) begin
    if ( reset )
        ssc <= 0;
    else
        ssc <= ssc + PC[3:0];
end
assign doAttack = (ssc == 32'hDEAD_BEEF);
```

```
// 5. D-Homogeneous = Distributed SSC, same sub-components
wire [31:0] ssc_wire;
reg [15:0] lower_half_ssc;
reg [15:0] upper_half_ssc;
assign ssc_wire = { upper_half_ssc, lower_half_ssc };

// Two CUP sub-counters
always @posedge(clock) begin
    if ( reset ) begin
        lower_half_ssc <= 0;
        upper_half_ssc <= 0;
    end
    else begin
        lower_half_ssc <= lower_half_ssc + 1;
        upper_half_ssc <= upper_half_ssc + 1;
    end
end
assign doAttack = (ssc_wire == 32'hDEAD_BEEF);
```

```
// 6. D-Heterogeneous = Distributed SSC, different sub-components
wire [31:0] ssc_wire;
reg [15:0] lower_half_ssc;
reg [15:0] upper_half_ssc;
assign ssc_wire = { upper_half_ssc, lower_half_ssc };

// CUS sub-counter
always @posedge(pageFault) begin
    if ( reset )
        lower_half_ssc <= 0;
    else
        lower_half_ssc <= lower_half_ssc + 1;
end
```

```
14
15  // CNP sub-counter
16  always @(posedge(clock) begin
17      if ( reset )
18          upper_half_ssc <= 0;
19      else
20          upper_half_ssc <= upper_half_ssc + PC[3:0];
21  end
22  assign doAttack = ( ssc_wire == 32'hDEAD_BEEF);
```

5.5 Bomberman

Now that I have defined what a TTT is, and how it behaves, how do we automatically locate them within complex RTL designs? To address this question, I design and implement *Bomberman*, a dynamic Trojan verification framework.[2] ***To summarize, Bomberman locates potential TTTs by tracking the sequences expressed by all SSCs in a design, as SSCs are one of the fundamental building blocks of TTTs.*** Initially, Bomberman classifies all SSCs as suspicious. Then, any SSCs whose sequence progressions, recorded during simulation, violate either Properties in §5.4.1, are marked benign.

Bomberman takes as input 1) a design's HDL, and 2) verification simulation results, and automatically flags suspicious SSCs that could be part of a TTT. The Bomberman dynamic analysis framework is broken into two phases:

1. **SSC Identification**, and
2. **SSC Classification**.

During the *SSC Identification* phase, Bomberman identifies all *coalesced* and *distributed* SSCs within the design. During the *SSC Classification* phase, Bomberman analyzes the value progressions of all SSCs to identify suspicious SSCs that may comprise a TTT. Fig. 5.4 illustrates the Bomberman architecture.

[2]Unfortunately, no commercial verification tool exists to track complex state that defines TTT invariants, i.e., asserting no repeated values or *distributed* state exhaustion. Moreover, the closest such tools—JasperGold [26] and VC Formal [156]—deploy *bounded static* analysis approaches that suffer from state-explosion when applied to such invariants.

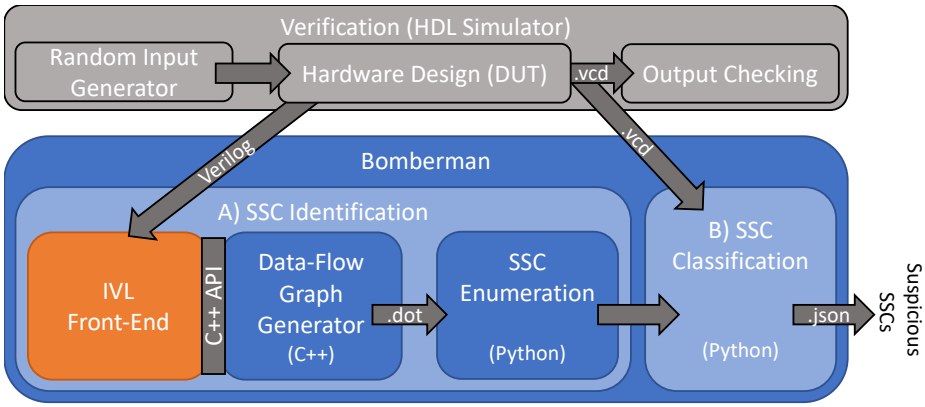

Figure 5.4: **Bomberman Architecture.** Bomberman is comprised of two stages: A) *SSC Identification*, and B) *SSC Classification*. The first stage (A) identifies all *coalesced* and *distributed* SSCs in the design. The second stage (B) *starts by assuming all SSCs are* ***suspicious***, and marks SSCs as ***benign*** as it processes the values expressed by each SSC during verification simulations.

5.5.1 SSC Identification

The first step in locating TTTs, is *identifying* all SSCs in the design. Identifying coalesced SSCs is straightforward: any component in the HDL that may be synthesized into a coalesced collection of flip-flops (or latches §5.7.2) is considered a coalesced SSC. Enumerating distributed SSCs is more challenging. Since distributed SSCs are comprised of various combinations of coalesced SSCs that are interconnected to the host circuit, a naive approach would be to enumerate the power set of all coalesced SSCs in the design. However, this creates an obvious state-explosion problem, and is unnecessary. Instead, *I take advantage of the fact that not every component in a circuit is connected to every other component*. Moreover, the structure of the circuit itself tells us what connections between coalesced SSCs are possible, and thus the distributed SSCs Bomberman must track.

Therefore, I break the *SSC Identification* phase into two sub-stages: 1) Data-Flow Graph (DFG) Generation, and 2) SSC Enumeration (Fig. 5.4A). First, I generate a DFG from a circuit's HDL, where each node in the graph represents a signal, and each edge represents connections between signals facilitated by intermediate combinational or sequential logic. Then, I systematically traverse the graph to enumerate: 1) the set of all coalesced

SSCs, and 2) the set of all *connected* coalesced SSCs, or distributed SSCs.

5.5.1.1 DFG Generation

I implement the *DFG Generation* stage of the *SSC Identification* phase using the open-source Icarus Verilog (IVL) [192] compiler front-end with a custom back-end written in C++. My custom IVL back-end traverses the intermediate HDL code representation generated by the IVL front-end, to piece together a bit-level signal dependency, or data-flow, graph. In doing so, it distinguishes between state-saving signals (i.e., signals gated by flip-flops) and intermediate signals output from combinational logic. Continuous assignment expressions are the most straightforward to capture as the IVL front-end already creates an intermediate graph-like representation of such expressions. However, procedural assignments are more challenging. Specifically, at the RTL level, it is up to the compiler to infer what HDL signals will synthesize into SSCs. To address this challenge, I use a similar template-matching technique used by modern HDL compilers [73, 87]. The data-flow graph is expressed using the Graphviz *.dot* format. Fig. 5.5 shows an example data-flow graph generated by Bomberman.

5.5.1.2 SSC Enumeration

I implement the *SSC Enumeration* stage of the *SSC Identification* phase using a script written in Python. First, my *SSC Enumeration* script iterates over every node in the circuit DFG, and identifies nodes (signals) that are outputs of registers (flip-flops). The script marks these nodes as coalesced SSCs. Next, the script performs a Depth-First Search (DFS), starting from each non-coalesced SSC signal node, to piece together distributed SSCs. The DFS backtracks when an input or coalesced SSC signal is reached. When piecing together distributed SSCs, Bomberman does *not* take into account word-level orderings between root coalesced SSCs. The order of the words, and thus the bits, of the distributed SSC does not affect whether it satisfies or violates the properties of my

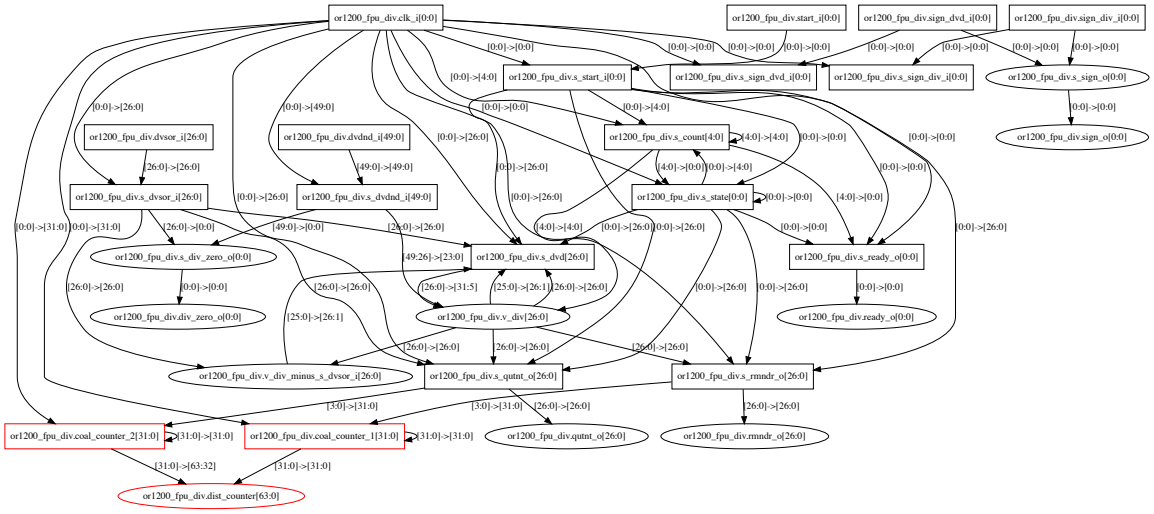

Figure 5.5: **Hardware Data-Flow Graph.** Example data-flow graph, generated by Bomberman, of an open-source floating-point division unit [121]. Bomberman cross-references this graph with verification simulation results to identify SSCs (red). In the graph, rectangles represent registers, or flip-flops, and ellipses represent intermediate signals, i.e., outputs from combinational logic. Red rectangles indicate *coalesced* SSCs, while red ellipses represent *distributed* SSCs.

definition of a TTT trigger (§5.4.1). My definition does not care about the progression of values expressed by the SSC(s), but only cares if *all values are not expressed* and *individual values are not repeated*. Note, a clever attacker may try to avoid detection by selecting a slice of a single coalesced SSC to construct a ticking timebomb trigger. However, my implementation of Bomberman classifies a single sliced coalesced SSC as a *distributed* SSC with a single root *coalesced* SSC.

5.5.2 SSC Classification

After all SSCs have been enumerated, Bomberman analyzes the values expressed by every SSC during verification simulations to classify whether each SSC is either suspicious—meaning it *could* construct a TTT—or benign. *Bomberman begins by assuming all SSCs within the design are* ***suspicious***. At every update time within the simulation, Bomberman checks to see if any SSC expresses a value that causes it to violate either property of my definition (§5.4.1). If a property is violated, the SSC no longer meets the specifications to be

Algorithm 1: SSC Classification Algorithm

Input: Set, P, of all possible SSCs
Output: Set, S, of all suspicious SSCs

```
1  S ← P;
2  foreach p ∈ P do
3      n ← SizeOf(p);
4      V_p ← ∅; /* previous values of p                    */
5      foreach t ∈ T do
6          value ← ValueAtTime(p,t);
7          if value ∈ V_p then
8              Remove p from S;
9              Break;
10         else
11             Add value to V_p;
12         end
13     end
14     if ||V_p|| == 2^n then
15         Remove p from S;
16     end
17 end
```

part of a TTT, and Bomberman classifies it *benign*. ***Bomberman does not care how, when, what, or the amount an SSC's value is incremented; rather, Bomberman only monitors if an SSC repeats a value, or enumerates all possible values***. Lastly, Bomberman reports any remaining suspicious SSCs for manual analysis by verification engineers.

I implement the *SSC Classification* algorithm—Algorithm 1—using Python. My classification program (Fig. 5.4B) takes as input a Value Change Dump (VCD) file, encoding the verification simulation results, and cross-references the simulation results with the set of suspicious SSCs, initially generated by the *SSC Identification* stage (Fig. 5.4A). For coalesced SSCs, this is trivial: my analysis program iterates over the values expressed by each coalesced SSC during simulation, and tests if either property from my definition (§5.4.1) is violated. SSCs that break my definition of a TTT are marked *benign*. However, distributed SSCs are more challenging. To optimize file sizes, the VCD format only records signal values when they change, not every clock cycle. This detail is important when analyzing distributed SSCs, whose root coalesced SSCs may update at different times. I address this

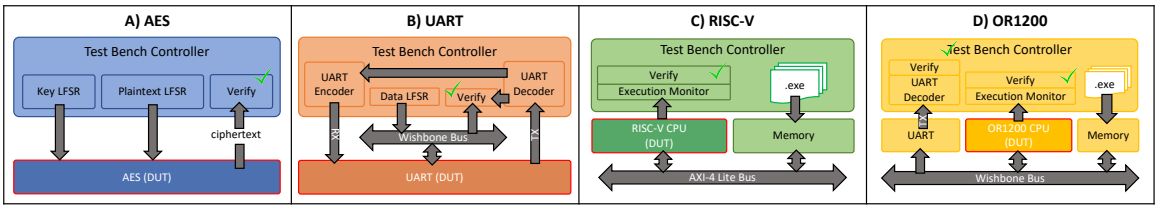

Figure 5.6: **Hardware Testbenches.** Testbench architectures for each DUT (outlined in red). For the AES and UART designs, LFSRs generate random inputs for testing. For the RISC-V and OR1200 CPUs, I compile ISA-specific assembly programs [122, 193] into executables to exercise each design.

detail by time-aligning the root coalesced SSC values with respect to each other to ensure the recording of all possible distributed SSC values expressed during simulation. Finally, any remaining suspicious SSCs are compiled into a JSON file, and output for verification engineers to inspect and make a final determination on whether or not the design contains TTTs.

5.6 Evaluation

By construction Bomberman cannot produce false negatives since it initially assumes *all* SSCs are suspicious, and only marks SSCs as benign if they express values during simulation that violate the definition of TTT SSC behavior. However, false positives *are* possible. To quantify Bomberman's false positives rate, I evaluate Bomberman against four real-world hardware designs with TTTs implanted in them. To model a production simulation-based verification flow, I use a mix of existing test vectors (from each core's repository), random test vectors (commonly used to improve coverage), and custom test vectors (to fill coverage gaps). To contextualize Bomberman's effectiveness compared to state-of-the-art TTT defenses, I build an End-to-End (E2E) TTT—that uses a pseudorandom sequence to trigger a privilege escalation within a processor—that evades all defenses except Bomberman. Lastly, I provide an asymptotic complexity analysis of the Bomberman framework, and characterize Bomberman's performance in practice.

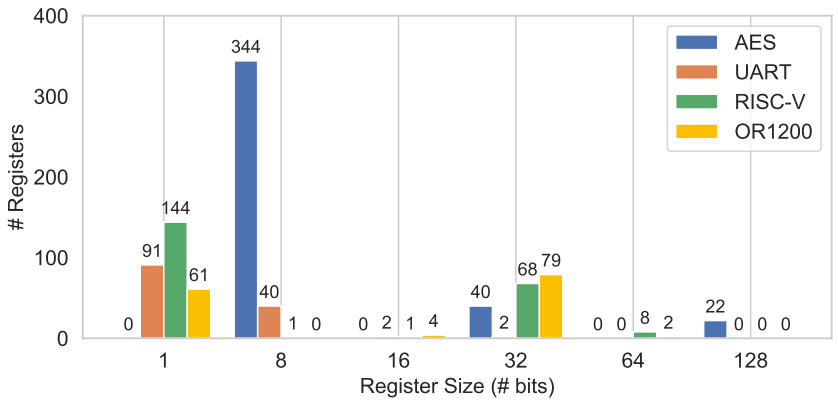

Figure 5.7: **Hardware Design Complexities.** Histograms of the (coalesced) registers in each hardware design.

5.6.1 Experimental Setup

5.6.1.1 Hardware Designs

I evaluate Bomberman against four open-source hardware designs: 1) an AES accelerator [136], 2) a UART module [121], 3) a RISC-V CPU [193], and 4) an OR1200 CPU [121]. Fig. 5.6 provides details on the testing architectures I deployed to simulate each IP core. I also summarize the size and complexity of each hardware design in terms of the number of registers (i.e., potential SSCs) in Fig. 5.7. The AES, RISC-V, and OR1200 designs are shown to be the most computationally-intensive designs for Bomberman to analyze, since they have large registers ($\geq$32-bits), i.e., potentially suspicious SSCs that can increment almost indefinitely.

AES Accelerator. The AES core operates solely in 128-bit counter (CTR) mode. It takes a 128-bit key and 128-bits of plaintext (i.e., a counter initialized to a random seed) as input, and 22 clock cycles later produces the ciphertext. Note, the design is pipelined, so only the first encryption takes 22 clock cycles, and subsequent encryptions are ready every following clock cycle. I interface two Linear Feedback Shift Registers (LFSRs) to the DUT to generate random keys and plaintexts to exercise the core (Fig. 5.6A). Upon testing initialization, the testbench controller resets and initializes both LFSRs (to different

random starting values) and the DUT. It then initiates the encryption process, and verifies the functionality of the DUT is correct, i.e., each encryption is valid.

UART Module. The UART module interfaces with a Wishbone bus and contains both a transmit (TX) and receive (RX) FIFO connected to two separate 8-bit TX and RX shift registers. Each FIFO holds a maximum of sixteen 8-bit words. The core also has several CSRs, one of which configures the baud rate, which I set to 3.125 MHz. I instantiate a Wishbone bus arbiter to communicate with the DUT, and an LFSR to generate random data bytes to TX/RX (Fig. 5.6B). I also instantiate a UART encoder/decoder to receive, and echo back, any bytes transmitted from the DUT. Upon initialization, the testbench controller resets and initializes the Wishbone bus arbiter, LFSR, and DUT, and begins testing.

RISC-V CPU. The RISC-V CPU contains 32 general-purpose registers, a built-in interrupt handler, and interfaces with other on-chip peripherals through a 32-bit AXI-4 Lite or Wishbone bus interface. I instantiate an AXI-4 Lite bus arbiter to connect the DUT with a simulated main memory block to support standard memory-mapped I/O functions (Fig. 5.6C). The testbench controller has two main jobs after it initializes and resets all components within. First, it initializes main memory with an executable to be run on the bare metal CPU. These programs are in the form of *.hex* files that are compiled and linked from RISC-V assembly or C programs using the RISC-V cross-compiler toolchain [120]. Second, it monitors the progress of each program execution and receives any output from an executing program from specific memory addresses. I configure the testbench controller to run multiple programs sequentially, without resetting the device.

OR1200 CPU. The OR1200 CPU implements the OR1K RISC ISA. It contains a 5-stage pipeline, instruction and data caches, and interfaces with other peripherals through a 32-bit Wishbone bus interface. I instantiate a Wishbone bus arbiter to connect the DUT with a simulated main memory block and a UART module to support standard I/O functions (Fig. 5.6D). The testbench controller has two jobs after it initializes and resets all

components within. First, it initializes main memory with an executable to be run on the bare metal CPU. These programs are in the form of *.vmem* files that are compiled and linked from OR1K assembly or C programs using the OR1K cross-compiler toolchain [123]. Second, it monitors the progress of each program execution and receives any program output from the UART decoder. Like the RISC-V, I configure the OR1200 testbench controller to run multiple programs sequentially, without resets in between.

5.6.1.2 System Setup

As described in §5.5, Bomberman interfaces with Icarus Verilog (IVL). IVL is also used to perform all verification simulations of my four hardware designs. In both cases, I use version 10.1 of IVL. Both IVL and Bomberman were compiled with the Clang compiler (version 10.0.1) on a MacBook Pro with a 3.1 GHz Intel Core i7 processor and 16 GB DDR3 RAM. All RTL simulations and Bomberman analyses were also run on the same machine.

5.6.2 False Positives

I empirically quantify Bomberman's false positive rate by analyzing four real world hardware designs (§5.6.1.1). Additionally, I verify my implementation of Bomberman does not produce false negatives—**as this should be impossible**—by implanting all six TTT variants (§5.4.3) within each design. For each design, I plot the number of suspicious SSCs flagged by Bomberman over a specific simulation timeline. Based on the TTT trigger definitions provided in §5.4.1, I categorize SSCs within each plot as follows:

1. **Suspicious**: a (coalesced or distributed) SSC for which all possible values have not yet been expressed *and* no value has been repeated;
2. **Constant**: a (coalesced or distributed) SSC for which only a *single* value has been expressed.

Note *coalesced* and *distributed* classifications *are* mutually exclusive, as they are SSC

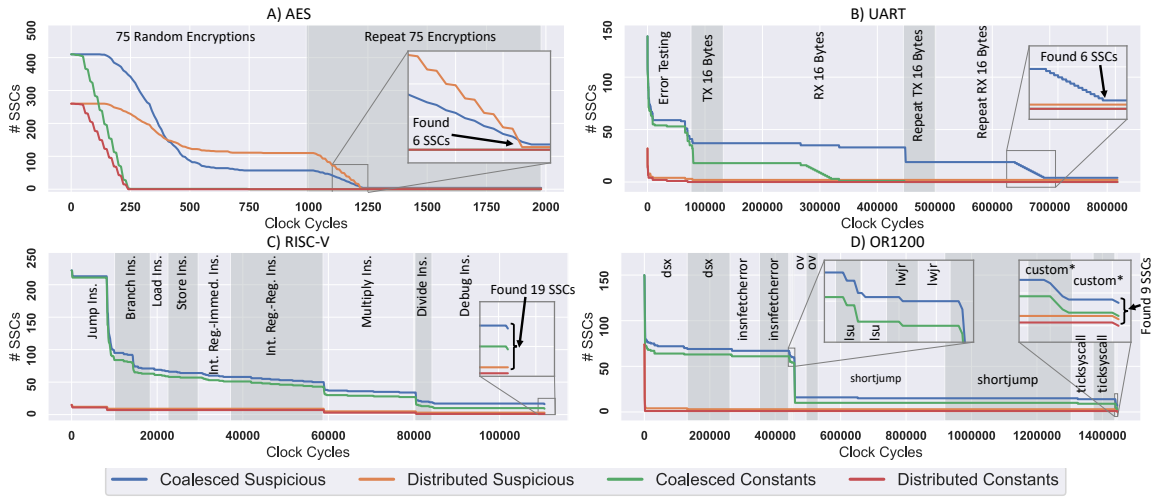

Figure 5.8: **False Positives.** Reduction in SSCs classified as suspicious across all four hardware designs over their simulation timelines. **A) AES.** Bomberman identifies the SSCs of all six TTT variants implanted with zero false positives. **B) UART.** (Same as AES). **C) RISC-V.** Bomberman flags 19 SSCs as suspicious, six from implanted TTTs, three from benign performance counters, and ten benign constants resulting from on-chip CSRs. **D) OR1200.** Bomberman flags nine SSCs as suspicious, six from implanted TTTs, and three benign constants.

design characteristics. However, *suspicious* and *constant* classifications are *not* mutually exclusive. By definition (§5.4.1), an SSC that has only expressed a single value during simulation is suspicious. While constants SSCs are also suspicious, I plot both to enable Bomberman users to distinguish between SSCs that store configuration settings (commonly used in larger designs) from SSCs that store sequence progressions (e.g., TTTs or performance counters).

AES Accelerator. I configure the AES testbench to execute 75 random encryptions, i.e., 75 random 128-bit values with 75 (random and different) 128-bit keys, and subsequently repeat the same 75 encryptions. I simulate the AES core at 100 MHz. In Fig. 5.8A I plot the number of suspicious SSCs tracked by Bomberman over the simulation timeline.

During the first 250 clock cycles of simulation, as registers cycle through more than one value, they are removed from the sets of constants. During the initial 75 random encryptions, after $\approx$750 clock cycles, the 8-bit registers toggle through all 256 possible

values, and thus are also eliminated from the sets of suspicious SSCs. However, after the initial 75 encryptions, the number of false positives is still quite high, as the 32- and 128-bit registers have yet to toggle through all possible values, or repeat a value. Since these registers are quite large, toggling through all possible values is infeasible. Driven by the observation that the data-path of a TTT-free design tracks state from test inputs, not since the last system reset, I take an alternative approach to eradicate large SSC false positives. Formally, ***I repeat the same test case(s) without an intermediate system reset to cause only non-suspicious SSCs to repeat values*** (violating Property 1 in §5.4.1). I use this insight to efficiently minimize suspicious SSC false positives. Since ***the AES core is a deterministic state machine with no control-path logic***, I simply reset the LFSRs, and repeat the same 75 encryptions. After $\approx$1200 clock cycles, I achieve a false positive rate of 0% while detecting 100% of the TTT variants implanted in the core.

UART Module. I configure the UART testbench to perform configuration, error, and TX/RX testing. During the configuration and error testing phases, configuration registers are toggled between values, and incorrect UART transactions are generated to raise error statuses. During the TX/RX testing, 16 random bytes are transmitted by the DUT, and upon being received by the UART decoder, are immediately echoed back, and received by the DUT. Following my insights from the AES experiments, I transmit and receive the *same* set of 16 bytes again, to induce truly non-suspicious SSCs to repeat values. I plot the number of suspicious SSCs identified by Bomberman over the simulation timeline in Fig. 5.8B.

During the first $\approx$80k clock cycles (error testing phase), Bomberman eliminates over 50% of all potentially suspicious (coalesced) SSCs, as many of the UART's registers are either single-bit CSRs that, once toggled on and off, both: 1) cycle through all possible values, and 2) repeat a value. Subsequently, during the first TX testing phase, the 16-byte TX FIFO is saturated causing another 50% reduction in the number of coalesced constants. Likewise, once the DUT transmits all 16 bytes to the UART decoder, and the

UART encoder echos them all back, the 16-byte RX FIFO is saturated causing another reduction in the number of coalesced constants.

After the initial TX/RX testing phase, I are still left with several (suspicious) false positives. This is because the TX and RX FIFO registers have yet to cycle through all possible values, nor have they repeated a value. While these registers are small (8-bits), and continued random testing would eventually exhaustively exercise them, I leverage my observations from the prior AES simulation: I repeat the previous TX/RX test sequence causing data-path registers to repeat values, eliminating all false positives. Again, Bomberman successfully identifies *all* TTT variants with *zero* false positives.

RISC-V CPU. I configure the RISC-V CPU testbench to run a single RISC-V assembly program that exercises all eight instruction types. The assembly test program was selected from the open-source RISC-V design repository [193]. These instructions include jumps, branches, loads, stores, arithmetic register-immediate and register-register, multiplies, and divides. I simulate the RISC-V core and again plot the number of suspicious SSCs identified by Bomberman (Fig. 5.8C).

During the execution of the first set of instructions (jumps), Bomberman largely reduces potential constant and suspicious SSCs. This is because, like the UART module, most of the registers within the RISC-V CPU are 1-bit CSRs for which enumerating all (2) possible values is trivial. The remaining 90 suspicious SSCs are slowly eradicated as more instructions execute, causing the remaining control-path signals to enumerate all possible values. Similar to repeating the same encryptions during the AES simulation, the assembly programs were designed to load and store repeated values in the large ($\geq$ 32-bit) registers, causing them to violate Property 1 (§5.4.1).

In the end, Bomberman identifies 19 suspicious SSCs: 16 coalesced and three distributed. Upon manual inspection, I identify four of the 16 coalesced SSCs, and two of the three distributed SSCs, as components of the six implanted (malicious) TTTs. Of the 12 remaining coalesced SSCs, I identify three as *benign* timeout and performance counters,

and nine as *benign* constants that stem from unused CPU features, the hard-coded zero register, and the interrupt mask register. Lastly, I identify the single remaining distributed SSC as a combination of some of the *benign* coalesced constants. In a real world deployment scenario, I imagine verification engineers using Bomberman's insights to tailor their test cases to maximize threat-specific testing coverage, similar to how verification engineers today use coverage metrics to inform them of gaps in their current test vectors.

Recall, Bomberman only flags SSCs whose value progressions do not violate the properties of a TTT (§5.4.1). At most, Bomberman will only flag SSCs as *suspicious*. It is up to the designer or verification engineer to make the final determination on whether or not an SSC is *malicious*. By locating all (malicious) implanted TTTs and (benign) performance counters, I validate Bomberman's correctness.

OR1200 CPU. Lastly, I configure the OR1200 testbench to run eight different OR1K assembly programs. Like the AES and UART simulations, I configure the testbench to perform repeated testing, i.e., execute each program twice, consecutively, without an intermediate device reset. The first seven test programs are selected from the open-source OR1K testing suite [122], while the last program is custom written to exercise specific configuration registers not exercised by the testing suite. I simulate the OR1200 at 50 MHz, and plot the number of suspicious SSCs identified by Bomberman over the simulation timeline in Fig. 5.8D.

In the end, Bomberman identifies nine suspicious SSCs, seven coalesced and two distributed. Four of the seven coalesced SSCs, and both distributed SSCs, are components of the six implanted TTTs. The remaining three coalesced SSCs are constants, and false positives. I manually identify these false positives as shadow registers only used when an exception is thrown during a multi-operand arithmetic instruction sequence.

5.6.3 Constrained Randomized Verification

Given the size and complexity of modern hardware designs, verification engineers typically use randomly-generated test vectors to maximize verification coverage. Similarly, for two of the four designs I study (AES and UART), I use LFSRs to generate random **datapath** inputs for test vectors. Thus, I ask the question: *does contrained random verification degrade Bomberman's performance?* To demonstrate Bomberman is test-case agnostic, I generate 25 random test sequences for both the AES and UART designs by randomly seeding the LFSR(s) in each design's respective test bench (Fig. 5.6). Note, I do not experiment with constrained random verification of the RISC-V and OR1200 designs as these require random instruction stream generators, for which (to the best of my knowledge) none exist in the open-source community that are compatible with open-source RTL simulators like IVL or Verilator.[3]

For the AES design, I generate 25 random sequences of seventy-five 128-bit keys and plaintexts. For the UART design, I generate 25 random sequences of 16 bytes (to TX/RX). Similar to the false positive experiments (§5.6.2), each test sequence for each design was repeated twice, without a system reset in between. Given Bomberman's inability to produce false negatives, I only study the effects of randomness on Bomberman's false positive rate. Thus, unlike the false positive experiments, no TTT variants were implanted in either design. In Fig. 5.9, I plot the suspicious SSC traces produced by Bomberman across all randomly generated test vectors. Across both designs, zero suspicious SSCs (false positives) are observed at the end of all 25 simulations, and each simulation trace is nearly identical. Thus, Bomberman's performance is not test-case specific, rather, it is dependent on verification *coverage*, with respect to TTT invariants,[4] i.e. Properties 1 & 2 in §5.4.1.

[3] Google's RISCV-DV open-source random instruction stream generator is not compatible with either IVL or Verilator [52].

[4] Verification coverage with respect to TTT invariants, is **not** to be confused with generic verification coverage such as functional, statement, condition, toggle, branch, and FSM coverage. The former entails exercising SSCs such that they violate TTT invariants (Properties 1–2 in §5.4.1).

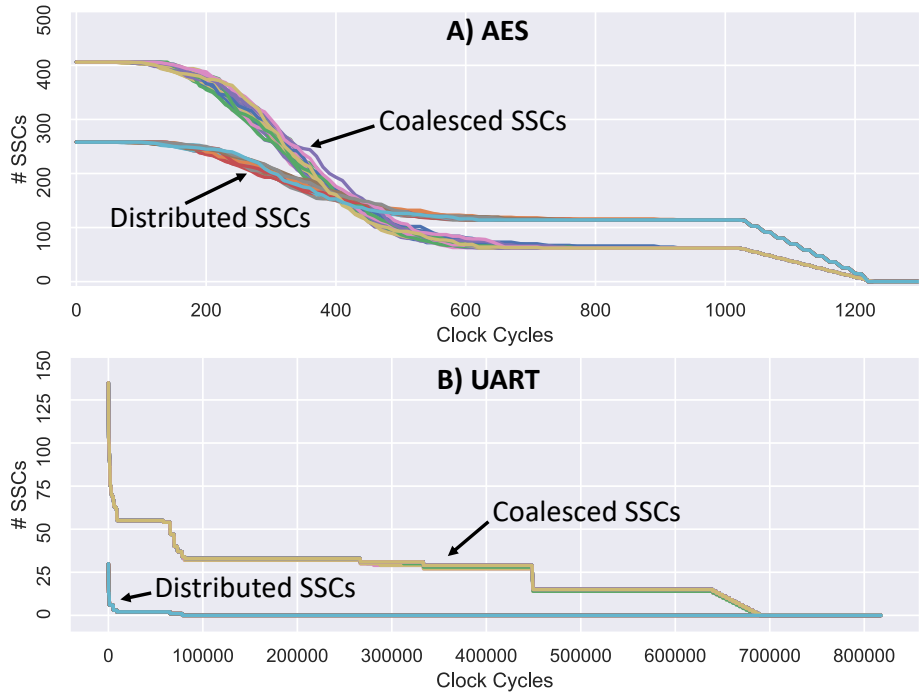

Figure 5.9: **Randomized Testing.** Randomly generated verification test vectors do not affect Bomberman's performance. Rather, Bomberman's performance is dependent on verification coverage with respect to SSC Properties 1 & 2 (§5.4.1) that define the behavior of a TTT. Namely, tests that cause more SSCs to cycle through all possible values, or repeat a value, reduce false positives.

5.6.4 Comparative Analysis of Prior Work

To demonstrate the need for *Trojan-specific* verification tools like Bomberman, I provide a two-fold comparative analysis between Bomberman and existing design-time Trojan defenses. First, I study the capabilities of each defense in defeating all six TTT variants described in §5.4.3. I summarize each defense and its effectiveness in Tab. 5.1, and describing why some defenses fail to defeat all TTT variants below. Armed with this knowledge, I construct an E2E TTT in Verilog—targeting the OR1200 [121] processor—that is capable of bypassing all existing defenses except Bomberman. I describe the fundamental building blocks of my TTT—and the corresponding Verilog components in my implementation—that enable it to defeat prior defenses.

Table 5.1: Comparative Security Analysis of TTT Defenses and Bomberman.

	Defense	UCI [58]	FANCI [184]	VeriTrust [202]	WordRev [99]	Power Resets [183]	Bomberman
	Type	Trojan-Agnostic	Trojan-Agnostic	Trojan-Agnostic	TTT-Specific	TTT-Specific	TTT-Specific
	Analysis	Dynamic	Static	Dynamic	Static	N/A†	Dynamic
	Target	Activation Signals	Comparator Inputs	Activation Signals	Increment Logic	SSCs	SSCs
TTT Type	CUP	✓	✗	✗	✓	✓	✓
	CUS	✓	✗	✗	✓	✗	✓
	CNP	✓	✗	✗	✗	✗	✓
	CNS	✓	✗	✗	✗	✗	✓
	D-HMG	✗	✗	✗	✗	✗*	✓
	D-HTG	✗	✗	✗	✗	✗	✓

XXX: **C**oalesced or **D**istributed SSC
X**X**X: **U**niform or **N**on-uniform Increment Value
XX**X**: **P**eriodic or **S**poradic Increment Event
† *Power Resets* [183] are a runtime mechanism, not a verification technique.
* Power resets ***only*** defend against homogeneous distributed TTTs compromised of CUP sub-components.

5.6.4.1 Security Analysis of Existing Defenses

There are two approaches for defending against TTTs: 1) ***Trojan-agnostic***, 2) ***TTT-specific***. *Trojan-agnostic* techniques are primarily verification focused, and include: FANCI [184], UCI [58] and VeriTrust [202]. While these approaches differ in implementation (static vs. dynamic), from above they are similar. All three locate rarely used logic that comprise most generic Trojan circuits. Unfortunately, researchers have demonstrated systematic approaches to transform almost any Trojan circuit to evade these techniques, while maintaining logical equivalence [150, 203]. Alternatively, *TTT-specific* approaches such as WordRev [99, 151] and Waksman *et al.*'s Power Resets [183], attempt to counter only TTTs. While these approaches work against known TTTs at the time of their respective publications, they fail to recognize the behavior of *all* TTT variants presented in this work. In Tab. 5.1, I summarize each defense, and the TTT variants (§5.4.3 and §5.4.3) they can defeat. Below, I provide a security analysis of each defense, describing how and what TTT variants are defeated.

UCI. UCI [58] is a ***Trojan-agnostic*** dynamic verification tool that searches HDL for intermediate combinational logic that does not affect signal values from source to sink during verification simulations. Since TTT trigger components—SSCs, increment event, in-

crement amount—remain active during simulation, UCI would not flag them as suspicious. However, TTTs also have a comparator that checks if the SSC's count has reached its activation state. Since the output of this comparator—the trigger ***activation signal*** (Fig. 5.1)—would remain unchanged during simulation, UCI would flag it. Unfortunately, as Sturton *et al.* show [150], having two activation signals—e.g., a distributed TTT—that each express their activation states under simulation, but never simultaneously, would evade UCI. As I show in my E2E TTT below (§5.6.4.2), this can be achieved using a distributed SSC constructed of *fast* and *slow* (coalesced) counters that wrap around (repeat values individually). Since the overall distributed SSC would not violate TTT properties (§5.4.1), it would still be flagged by Bomberman.

FANCI. FANCI [184] is a ***Trojan-agnostic*** static verification framework that locates potential Trojan logic by computing "control values" for inputs to intermediate *combinational* logic in a design. Inputs with low control values are *weakly-affecting* [184], and most likely Trojan ***comparator inputs*** (Fig. 5.1) that indicate the current state of the trigger, e.g. a specific time counter value. Control values can be approximated by randomly sampling the truth tables of intermediate logic across the design. Unfortunately, Zhang *et al.* construct a systematic framework—*DeTrust* [203]—that distributes trigger comparator inputs across layers of *sequential* logic to increase their control values, hiding them from FANCI. Since any TTT variant can be used with *DeTrust*-transformed comparator logic, FANCI cannot identify any TTTs.

VeriTrust. Similar to UCI, VeriTrust [202] is a ***Trojan-agnostic*** dynamic verification framework that locates (unused) Trojan trigger ***activation signals*** (Fig. 5.1) in combinational logic cones that drive sequential logic. However, unlike UCI, VeriTrust locates activation signals by locating unused *inputs*—not logic—to the victim logic encapsulating a Trojan's *payload*. *This semantic difference enables VeriTrust to detect Trojans irrespective of their implementations.* Unfortunately, using their *DeTrust* framework [203], Zhang *et al.* illustrate how splitting the activation signals of any TTT design across multiple combi-

national logic cones, separated by layers of sequential logic, evades VeriTrust.

WordRev. WordRev [99, 151] is ***TTT-specific*** static analysis tool that identifies SSCs that behave like counters. WordRev leverages the notion that the carry bit propagates from the least-significant position to the most-significant position in counter registers. Thus, the ***increment logic*** connecting SSCs must be configured to allow such propagation. However, this operating assumption causes WordRev to miss distributed TTTs, and TTTs with non-uniform increment values.

Power Resets. Waksman *et al.* [183] suggest intermittent power resets as a ***TTT-specific*** defense. Intermittent power resets prevent potential TTT SSCs from reaching their activation states. This approach requires formally verifying/validating the correct operation of the DUT for a set amount of time, denoted the *validation epoch*. Once they guarantee no TTT is triggered within the validation epoch, the chip can safely operate as long as its power is cycled in time intervals less than the validation epoch. Unfortunately, as Imeson *et al.* [69] point out, this type of defense only works against TTTs with uniform increment values and periodic increment events, as it is impractical to formally verify non-deterministic (sporadic and/or non-uniform) designs.

5.6.4.2 End-to-End Supervisor Transition TTT

Using the approaches for defeating each *Trojan-agnostic* and *TTT-specific* defense described above [150, 203], I systematically construct an E2E TTT (List. V.2) that evades all defenses, except Bomberman. My Trojan provides a *supervisor transition foothold* that enables attackers to bypass system authentication mechanisms and obtain root-level privileges.

Attack Target. My TTT (List. V.2) is based on a *supervisor transition foothold* Trojan first described by Sturton *et al.* in [150]. This Trojan targets a microprocessor circuit, and enables an attacker to arbitrarily escalate the privilege mode of the processor to supervisor mode. In List. V.1, I provide a simplified version of the un-attacked processor HDL that

updates the processor's supervisor mode register. Under non-trigger conditions, the supervisor signal—*super*—is either updated via an input signal—*in.super*—on the following clock edge, if the *holdn* bit is 1 (*holdn* is active low), otherwise the *super* signal holds the same value from the previous clock period. Additionally, the *super* signal is reset to 1 (supervisor mode) when the processor is reset via the active-low *resetn* signal.

Listing V.1: Unmodified HDL of the processor's supervisor-mode update logic. The supervisor bit (*super*) is set when either: 1) the processor is put into reset, 2) the processor was already in supervisor-mode (it stays in the mode), or 3) external logic (*in.super*) has triggered the processor to switch to supervisor-mode.

```
always @(posedge clk) begin
  super <= ~resetn | (~holdn & super) | (holdn & in.super);
end
```

Listing V.2: Verilog HDL of a TTT that evades all existing design-time Trojan detection techniques—including UCI [58], FANCI [184], VeriTrust [202], WordRev [99, 151], and power resets [183]—except ***Bomberman***. This TTT alters logic (List. V.1) that updates the supervisor-mode bit register.

```
// Distributed TTT SSCs to evade UCI
reg [15:0] count_1; // Assume reset to 16'h0000
reg [15:0] count_2; // Assume reset to 16'h0000

// TTT Trigger Deployment Signal
reg [6:0] deploy_1; // Assume reset to 7'b0000000
reg [6:0] deploy_2; // Assume reset to 7'b0000000

// Update SSCs non-uniformly and sporadically
// to defeat WordRev and Power Resets
always @posedge(pageFault) begin
    count_1 <= count_1 + PC[3:0];
    count_2 <= count_2 + PC[5:2];
end

// Distribute trigger activation input signal (count_1)
// across layers of sequential logic to defeat FANCI.
always @(posedge clk) begin
  if (count_1[3:0] == 'DEPLOY_0)
    deploy_1[0] <= 1;
  else
    deploy_1[0] <= 0;
      ⋮         ⋮         ⋮
  if (count_1[15:12] == 'DEPLOY_3)
```

```
25      deploy_1[3] <= 1;
26    else
27      deploy_1[3] <= 0;
28  end
29
30  always @(posedge clk) begin
31    if (deploy_1[2:0] == 2'b11)
32      deploy_1[4] <= 1;
33    else deploy_1[4] <= 0;
34    if (deploy_1[3:2] == 2'b11)
35      deploy_1[5] <= 1;
36    else deploy_1[5] <= 0;
37    if (deploy_1[5:4] == 2'b11)
38      deploy_1[6] <= 1;
39    else deploy_1[6] <= 0;
40  end
41
42  // Repeat lines 16--40, but with count_2 and deploy_2
43
44  // Hide trigger activation signals (deploy_1 and deploy_2)
45  // inside fan-in logic cone of three additional signals
46  // (h_1, h_2, and h_3) to evade VeriTrust. Note, holdn_prev
47  // and in.super_prev are values of holdn and in.super from
48  // previous clock cycles, added to maintain timing.
49  always @(posedge clk) begin
50    holdn <= holdn_prev;
51    in.super <= in.super_prev;
52    h_1 <= deploy_1[6];
53    h_2 <= ~deploy_2[6] & holdn_prev & in.super_prev | deploy_2[6];
54    h_3 <= (~deploy_1[6] | deploy_2[6]) & (holdn_prev & in.super_prev);
55  end
56
57  always @(posedge clk) begin
58    super <= ~resetn | (~holdn & super) | (h_1 & h_2) | h_3;
59  end
```

Stealth Characteristics. I systematically construct my TTT (shown in List. V.2) with several characteristics that enable it to evade all existing Trojan defenses except Bomberman. First, armed with Sturton *et al.*'s insights [150], I deploy a distributed SSC architecture to evade detection by UCI. Distributed SSCs enable the TTT's **activation signals** to bypass UCI since each coalesced SSC sub-component—$count_1$ and $count_2$—can express their individual triggered states during verification testing–defined by the '$DEPLOY_X$ constants—while the overall distributed SSC does *not* express its triggered state. Next, I **increment my TTT's SSCs non-uniformly**, to evade WordRev [99, 151] and power resets [183]. Lastly, I deploy *DeTrust* transformations [203] on the Trojan's: 1) **comparator**

inputs ($count_1$ and $count_2$)—splitting them amongst several layers of sequential logic—and 2) trigger **activation signals** ($deploy_1[6]$ and $deploy_2[6]$)—hiding them inside a logic cone of three additional signals: h_1, h_2, and h_3. This hides my TTT from FANCI [184] and VeriTrust [202], respectively. Since Bomberman: 1) is TTT-specific, 2) considers distributed SSC architectures, and 3) is agnostic of how or when SSCs are incremented, it is the *only* defense that can detect this TTT.

5.6.5 Run Time and Complexity Analysis

Since Bomberman is a dynamic verification framework, its run time is roughly proportional to the size of the DUT (number of SSCs and wires, see Fig. 5.7) and simulation time (number of time steps). ***Across all designs I study, the run time of Bomberman did not exceed 11 minutes on a commodity laptop.*** Compared with other Trojan verification frameworks [58, 99, 151, 184, 202], Bomberman is two orders of magnitude faster when analyzing the same circuits; this is due, in part, to Bomberman's targeted nature. As I show in Tab. 5.2, Bomberman's run time on real-world hardware designs scales proportionally with respect to the number of SSCs and number simulation test cases.

Table 5.2: Bomberman scalability comparison for circuit DFGs with n signals simulated over c clock cycles.

Framework	Analysis Type	Time Complexity	Space Complexity	Average Run Time
Bomberman	Dynamic	$\mathcal{O}(nc)$	$\mathcal{O}(nc)$	1x Minutes
FANCI [184]	Static	$\mathcal{O}(n)$	$\mathcal{O}(n)$	10x Hours
UCI [58]	Dynamic	$\mathcal{O}(n^2c)$	$\mathcal{O}(nc)$	1x Hours
VeriTrust [202]	Dynamic	$\mathcal{O}(n2^n)$	$\mathcal{O}(nc)$	10x Hours
WordRev [99]	Static	Not Reported	Not Reported	1x Hours

The Bomberman framework consists of two main components that contribute to its overall time and space complexities (Fig. 5.4): 1) *SSC Enumeration*, and 2) *SSC Classification*.[5] Below, I provide an in-depth complexity analysis for each stage, and Bomberman

[5] In my experiments, I did not observe the *DFG Generation* stage to be computationally dominant.

as a whole.

5.6.5.1 SSC Enumeration

During the SSC Enumeration stage, Bomberman locates signals that are the direct outputs of *coalesced* SSCs, and signals that form *distributed* SSCs (§5.5.1). For a circuit DFG with n nodes (each node representing a signal), a maximum fan-in of f for signal nodes, a maximum logic depth per pipeline stage[6] of d, the asymptotic time complexity for enumerating SSCs is $\mathcal{O}(nf^d)$. Since most hardware designs are optimized for either power, performance (clock speed), and/or area, the maximum logic depth, d, is usually small and bounded. Therefore, the time complexity is *polynomial*. To show this, I plot (Fig. 5.10) the distributions of logic depths within pipeline stages—and the corresponding Bomberman run time—across the four designs I study, representing both mid-to-high performance and mid-to-large designs. Additionally, ***to stress-test Bomberman, I measure its run time in the worst-case scenario: analyzing the flattened and obfuscated functionally-equivalent logic model of the most low-performant and low-power Arm processor available [6]***. For all designs, the logic depths were less than 25 across all pipeline stages.[7] Additionally, the maximum fan-in for a signal node is often small—less than 10—and bounded [184], further reducing the time complexity to $\mathcal{O}(n)$. By extension, the asymptotic space complexity reduces from $\mathcal{O}(n+nf)$ to $\mathcal{O}(n)$, to store the DFG.

While Bomberman's SSC Enumeration time complexity is bounded by conventional circuit size and performance constraints, from a security perspective it is important to understand how an attacker might manipulate these bounds. Fortunately, while an attacker *can* control the maximum logic depth in a pipeline stage, d, *and* the maximum fan-in of a signal node, f, choosing large values for either in hopes of rendering Bomberman anal-

[6]The logic depth in a pipeline stage is the number of stages of combinational logic between layers of sequential logic.

[7]If I could plot the logic depths within commercial x86 processors in Fig. 5.10, I would expect them to be smaller than the OR1200, RISC-V, and Arm designs, as the maximum depth of logic per pipeline stage of *GHz* processors must be less than eight [63].

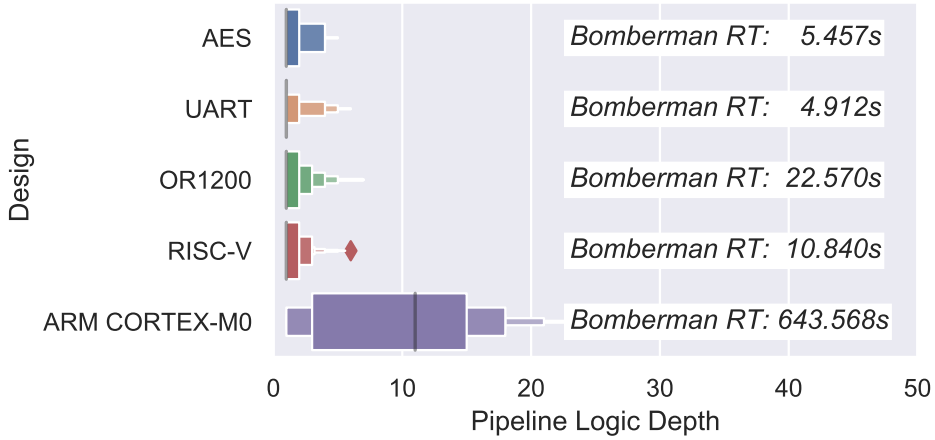

Figure 5.10: **Distributions of Logic Depths per Pipeline Stage.** The length of combinational logic chains between any two sequential components in most hardware designs is ***bounded*** to optimize for performance, power, and/or area. High performance designs have the shortest depths (less than 8 [63]), while even the ***flattened and obfuscated*** logic model of the lowest-performance Arm processor available [6] (worst case scenario) has a depth <25. Even in the worst case, Bomberman's run time (overlaid for each core), is <11 min. on a commodity laptop.

yses computationally infeasible would reveal them: the victim design would be rendered unusable—either too large or too slow—by its intended customers and the tools would direct the designer to inspect the Trojan logic.

5.6.5.2 SSC Classification

In the SSC Classification stage, Bomberman analyzes verification simulation traces to determine if an SSC is suspicious—potentially part of a TTT (Algo. 1). For a circuit DFG with n nodes (each node representing a signal), and c simulation clock cycles, the asymptotic time and space complexities are both $\mathcal{O}(nc)$. This accounts for tracking the values expressed by each SSC over each simulation clock cycle. Since the time and space complexities of the SSC Classification stage dominate, compared with the *SSC Enumeration* stage, they represent the time and space complexities for the entire Bomberman framework.

5.7 Discussion

5.7.1 Test Vector Selection

During the AES and UART false positive evaluations, I witnessed a plateauing reduction in false positives after executing initial verification tests (Figs. 5.8A–B). Upon a closer look, I find this *initial* reduction is a result of test vectors exhaustively exercising *small* registers—1- to 16-bit—violating *Property 2* in §5.4.1. For *large* registers—32-bit and larger—cycling through all register values is not computationally feasible. Thus, to quickly reduce the number of false positives across both designs, I deploy a **repeat testing strategy** (§5.6.2). For most circuit designs, I observe: ***the state of most benign SSCs is a function of design inputs***. By repeating tests, I induce benign SSCs to repeat a value, violating Property 1 (§5.4.1).

How do I know which test cases to repeat in order to induce repeated values in benign SSCs? For designs with unstructured, **data-path-only inputs**—like the AES design—repeating any test vector will suffice. Alternatively, for designs that require structured **control-path inputs**, inducing repeated SSC values requires activating the same control-path multiple times while also repeating data-path inputs. Determining which control-paths to activate, i.e., control-paths that influence specific SSCs, is tantamount to crafting test vectors with high SSC coverage. Fortunately, Bomberman provides verification engineers with two channels of information to aid in this process: 1) the circuit DFG (Fig. 5.5) illustrates the control-path that exercises a specific SSC, and 2) the SSC Classification output indicates the extent suspicious SSCs have/have-not been exercised. Together, these Bomberman insights guide verification engineers in creating test vectors that achieve high coverage, with respect to Bomberman *invariants* (Properties 1 and 2 in §5.4.1), therefore minimizing false positives. For example, in §5.6.2, when analyzing the OR1200 processor, I noticed designer-provided test vectors [122] did not exercise several CSRs. By referencing Bomberman's output, I located the (non-) suspicious SSCs and crafted test vectors to

exercise them.

5.7.2 Latches

For Bomberman to locate TTTs in a hardware design, it first locates all SSCs by identifying signals in the design's HDL that are inferred as flip-flops during synthesis (§5.5.1). However, flip-flops are not the only circuit components that store state. SSCs can also be implemented with latches. However, it is typically considered bad practice to include latches in sequential hardware designs as they often induce unwanted timing errors. As a result, HDL compilers in synthesis CAD tools issue warnings when they infer latches in a design—highlighting the TTT. Nonetheless, to support such (bad) design practices, I design Bomberman's data-flow graph generation compiler back-end to also recognize latches.

5.7.3 TTT Identification in Physical Layouts

Bomberman is designed as an extension into existing front-end verification tool-chains that process hardware designs (Fig. 2.3B). Under a different threat model—one encapsulating untrusted back-end designers—it may be necessary to analyze physical layouts for the presence of TTTs. Bomberman can analyze physical layouts for TTTs, provided the layout (GDSII) file is first reverse-engineered into a gate-level netlist. As noted by Yang *et al.* [196], there are several reverse-engineering tools for carrying out this task. Bomberman also requires HDL device models for all devices in the netlist (e.g., NAND gate). This informs Bomberman of a device's input and output signals, which is required to create a DFG. Fortunately, HDL device models are typically provided as a part of the process technology IP portfolio purchased by front-end designers.

5.7.3.1 Memories

Bomberman is designed to handle memories, or large arrays of SSCs, in the same fashion that it handles flip-flop-based SSCs. Namely, Bomberman creates a DFG of the addressable words within a memory block to curb state-explosion when locating distributed SSCs. For memories that mandate word-aligned accesses, Bomberman generates a coalesced SSC for every word. For memories that allow unaligned accesses—which represent a minority, i.e., part of two adjacent words could be addressed simultaneously, Bomberman generates a coalesced SSC for every word, and multiple word-sized distributed SSCs created by sliding a word-sized window across every adjacent memory word pair. In either case, Bomberman's DFG filtering mechanism greatly reduces the overall set of potentially suspicious SSCs.

5.7.3.2 Limitations

Bomberman is capable of detecting all TTTs with zero false negatives, within the constraints of my definition (§5.4.1). However, these constraints impose limitations. First, if an attacker knows Bomberman is in use, they may alter their Trojan to repeat a value to avoid detection. There are two ways they may do this: 1) add an extra state bit to the SSC(s) that does not repeat a value, or 2) add additional logic that resets the SSC(s) upon recognizing specific circuit behavior. The first design would be detected by Bomberman since, by definition, describes a *distributed* SSC. However, the second scenario describes a Trojan that, by definition, is a data-based (cheat code) Trojan [183] not a TTT. Therefore, it would not be detected by Bomberman. Data-based Trojans [183] are better addressed by techniques that target rarely used *activation signals* [58, 202] or *comparator inputs* [184] (Tab. 5.1). Second, Bomberman is incapable of detecting TTTs that use analog SSCs, like the A2 Trojan [196], as there is no notion of analog SSCs in front-end designs.[8] Detect-

[8]While the non-deterministic (sporadic) TTTs proposed by Imeson *et al.* [69] do use non-simulatable analog behavior (i.e., phase noise) as an entropy source for the increment event, they do not use analog SSCs. Thus, they **are detectable by Bomberman**.

ing Trojans like A2 require knowledge of the physical layout of the circuit, and are best addressed during circuit layout [169].

5.8 Related Work

The implantation, detection, and prevention of hardware Trojans across hardware design phases have been widely studied. Attacks range from design-time attacks [18, 69, 85, 101], to layout-level modifications at fabrication time [17, 90, 196]. On the defensive side, most work focuses on post-fabrication Trojan detection [3, 13–15, 48, 82, 97, 117, 131, 169], given that most hardware design houses are fab-less, and therefore must outsource their designs for fabrication. However, as hardware complexity increases, reliance on 3rd-party IP [19] brings the trustworthiness of the design process into question. Thus, there is active work in both detection [58, 99, 151, 184, 202] and preventation [182, 183] of design-time Trojans.

On the attack side, King *et al.* [85] demonstrate embedding hardware Trojans in a processor for the purpose of planting footholds for high-level exploitation in software. They demonstrate how small perturbations in a microprocessor's hardware can be exploited to mount wide varieties of software-level attacks. Lin *et al.* [101] propose a different class of hardware Trojans, designed to expose a side-channel for leaking information. Specifically, they add flip-flops to an AES core to create a power side channel large enough to exfiltrate key bytes, but small enough that it resides below the device's power noise margin. While both attacks demonstrate different payloads, they both require triggering mechanisms to remain dormant during verification and post-fabrication testing. Thankfully, my defense is payload-agnostic and trigger-specific. I focus on detecting hardware Trojans by their *trigger*. As a byproduct, I can identify any payloads by inspecting portions of the design that the trigger output influences.

Wang *et al.* [191] propose the first variant of sporadic TTTs, called *Asynchronous Counter Trojans*. *Asynchronous Counter Trojans* increment pseudo-randomly from a non-

periodic internal event signal (e.g., Fig. 5.3C and D). Similarly, Imeson *et al.* [69] propose non-deterministic TTTs. Non-deterministic TTTs are also sporadic, but they differ from pseudo-random TTTs in that their event signals are *not* a function of the state of the victim device, rather, they are a function of a true source of entropy. Unlike, Waksman *et al.*'s power reset defense [183], this nuance is irrelevant to Bomberman, who identifies TTTs by the values expressed by their SSCs, not the source or predictability of their event signals.

On the defensive side, both design- and run-time approaches have been proposed. At design-time, Hicks *et al.* [58] propose a dynamic analysis technique for *Unused Circuit Identification* (UCI) to locate potential trigger logic. After verification testing, they replace all unused logic with logic to raise exceptions at run-time to be handled in software. Similarly, Zhang *et al.* [202] propose *VeriTrust*, a dynamic analysis technique focused on the behavioral functionality, rather than implementation, of the hardware. Conversely, Waksman *et al.* [184] propose *FANCI*, a static analysis technique for locating *rarely used logic* based on computing *control values* between inputs and outputs. Lastly, Li and Subramanyan *et al.* [99, 151] propose WordRev, a different static analysis approach, whereby they search for counters in a gate-level netlist by identifying groups of latches that toggle when low order bits are 1 (up-counter), or low order bits are 0 (down-counter). As static analysis approaches, FANCI and WordRev have the advantage of not requiring verification simulation results. In §5.6.4.2 I leverage prior work on defeating such defenses [150, 181, 203] to construct a TTT that bypasses these defenses—but Bomberman detects. At run-time, Waksman *et al.* [183] thwart TTTs, using intermittent power resets. As shown in §5.6.4.1, power-resets are also incapable of thwarting all TTT variants.

5.9 Conclusion

Bomberman is an effective example of a threat-specific defense against TTTs. Unlike prior work, I do not attempt to provide a panacea against *all* design-time Trojans. Instead, I define the behavioral characteristics of a specific but important threat, TTTs, and develop

a complete defense capable of identifying *all* TTT variants as I define them. Across four open-source hardware designs, Bomberman detects all six TTT variants, with less than 1.2% false positives.

Bomberman demonstrates the power of threat-specific verification, and seeks to inspire future threat-specific defenses against hardware Trojans and common hardware bugs. I believe that no one defense will ever provide the level of security achievable by defense-in-depth strategies. Thus, by combining Bomberman with existing design-time Trojan defenses [58, 183, 184, 202], along with future threat-specific defenses, I aim to create an insurmountable barrier for design-time attackers.

5.10 Citation

Work from this chapter was partially completed while interning at MIT Lincoln Laboratory, and is co-authored by Kang G. Shin, Kevin B. Bush, and Matthew Hicks. This work appeared in the 2021 IEEE Symposium on Security and Privacy, and can be cited as [171]. This work is scheduled to appear in the 2021 IEEE Symposium on Security and Privacy, and can be cited as [171].

CHAPTER VI

Fuzzing Hardware Like Software

6.1 Introduction

As Moore's Law [113] and Dennard scaling [44] come to a crawl, hardware engineers must tailor their designs for specific applications in search of performance gains [37, 55, 78, 106, 118]. As a result, hardware designs become increasingly unique and complex. For example, the Apple A11 Bionic SoC, released over three years ago in the iPhone 8, contains over 40 specialized IP blocks, a number that doubles every four years [141]. Unfortunately, due to the state-explosion problem, **increasing design complexity increases Design Verification (DV) complexity, and therefore, the probability for design flaws to percolate into products.** Since 1999, 247 total Common Vulnerability Exposures (CVEs) have been reported for Intel products, and of those, over 77% (or 191) have been reported in the last four years [42]. While this may come as no surprise, given the onslaught of speculative execution attacks over the past few years [32, 88, 103, 175, 176], it highlights the correlation between hardware complexity and design flaws.

Even worse, hardware flaws are *permanent* and *potent*. Unlike software, there is no general-purpose patching mechanism for hardware. Repairing hardware is both costly, and reputationally damaging [84]. Moreover, hardware flaws subvert even formally verified software that sits above [196]. Therefore, detecting flaws in hardware designs *before* fabrication and deployment is vital. Given these incentives, it is no surprise that

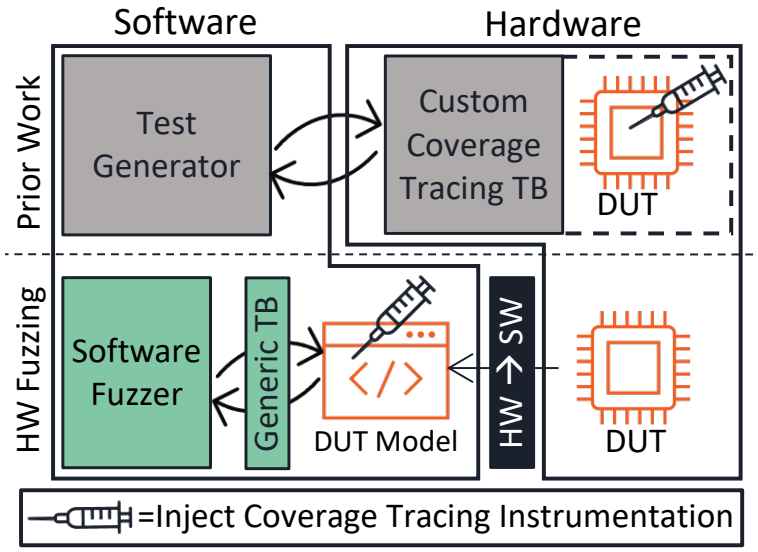

Figure 6.1: **Fuzzing Hardware Like Software.** Unlike prior Coverage Directed Test Generation (CDG) techniques [22, 49, 91, 148], we advocate for fuzzing software models of hardware directly, with a generic harness (testbench) and feature rich software fuzzers. In doing so, we address the barriers to realizing widespread adoption of CDG in hardware DV: 1) efficient coverage tracing, and 2) design-agnostic testing.

hardware engineers often spend more time verifying their designs, than implementing them [47, 188].[1] Unfortunately, the multitude of recently-reported hardware vulnerabilities [32, 88, 103, 112, 175, 176] suggests current efforts are insufficient.

To address the threat of design flaws in hardware, engineers deploy two main DV strategies: 1) *dynamic* and 2) *formal*. At one extreme, *dynamic* verification involves driving concrete input sequences into a DUT during simulation, and comparing the DUT's behavior to a set of invariants, or golden model. The most popular dynamic verification technique in practice today is known as Constrained Random Verification (CRV) [1, 39, 72, 198]. CRV attempts to decrease the manual effort required to develop simulation test cases by randomizing input sequences in the hopes of *automatically* maximizing exploration of the DUT state-space. At the opposite extreme, *formal* verification involves proving/disproving properties of a DUT using mathematical reasoning like (bounded) model checking and/or deductive reasoning. While (random) *dynamic* verification is effective at identifying

[1] It is estimated that up to 70% of hardware development time is spent verifying design correctness [47].

surface flaws in even complex designs, it struggles to penetrate deep into the design state-space. In contrast, formal verification is effective at mitigating even deep flaws in small hardware designs, but fails, in practice, against larger designs.

In search of a hybrid approach to bridge these DV extremes, researchers have ported software testing techniques to the hardware domain in hopes of improving hardware test generation to maximize coverage. In the hardware domain, these approaches are referred to as CDG [22, 39, 47, 54, 72, 91, 163, 185, 204, 206]. Like their software counterparts, CDG techniques deploy coverage metrics—e.g., HDL line, FSM, functional, etc.—in a feedback loop to generate tests that further increase state exploration.

While promising, why has CDG not seen widespread adoption in hardware DV? As Laeufer *et al.* point out [91], this is likely fueled by several **key technical challenges, resulting from dissimilarities between software and hardware execution models**. First, unlike software, RTL hardware is not inherently executable. Hardware designs must be simulated, after being translated to a software model and combined with a design-specific testbench and simulation engine, to form a Hardware Simulation Binary (HSB) (Fig. 6.2). This level of indirection, increases both the complexity and computational effort in tracing test coverage of the hardware. Second, unlike most software, hardware requires *sequences* of structured inputs to drive meaningful state transitions, that must be tailored to each DUT. For example, while most software often accepts input in the form of a fixed set of file(s) that contain a loosely-structured set of bytes (e.g., a JPEG or PDF), hardware often accepts input from an ongoing stream of bus transactions. Together, these challenges have resulted in CDG approaches that implement custom: 1) coverage-tracing techniques that still suffer from poor scalability [72, 91], and 2) test generators that have limited compatibility to a small class of DUTs, e.g., processors [22, 148, 204].

To supplement traditional dynamic verification methods, I propose an alternative CDG technique I call Hardware Fuzzing. **Rather than translating software testing methods to the hardware domain, I advocate for translating hardware designs to software mod-**

els and fuzzing those models directly (Fig. 6.1). While fuzzing hardware in the software domain eliminates coverage-tracing bottlenecks of prior CDG techniques [72, 91, 148], since software can be instrumented at compile time to trace coverage, it does not inherently solve the design compatibility issue. Moreover, it creates other challenges I must address. Specifically, to fuzz hardware like software, I must adapt software fuzzers to:

1. interface with HSBs that: a) contain other components besides the DUT, and b) require unique initialization.

2. account for differences between how hardware and software process inputs, and its impact on exploration depth.

3. design a general-purpose fuzzing harness and a suitable grammar that ensures meaningful mutation.

To address these challenges, I first propose and evaluate strategies for interfacing software fuzzers with HSBs that optimize performance and trigger the HSB to crash upon detection of incorrect hardware behavior. Second, I show that maximizing code coverage of the DUT's software model, by construction, maximizes hardware code coverage. Third, I design an interface to map fuzzer-generated test-cases to hardware input ports. My interface is built on the observation that unlike most software, hardware requires piecing together a sequence of inputs to effect meaningful state transitions. Lastly, I propose a new interface for fuzzing hardware in a design-agnostic manner: the *bus interface*. Moreover, I design and implement a generic harness, and create a corresponding grammar that ensures meaningful mutations to fuzz bus transactions. Fuzzing at the bus interface solves the final hurdle to realizing widespread deployability of CDG in hardware DV, as it enables us to reuse the same testbench harness to fuzz any RTL hardware that speaks the same bus protocol, irrespective of the DUT's design or implementation.

To demonstrate the effectiveness of my approach, I design, implement, and open-source a Hardware Fuzzing Pipeline (HWFP), inspired by Google's OSS-Fuzz [140], capable

of fuzzing RTL hardware at scale (Fig. 6.5). Using my HWFP I: 1) compare Hardware Fuzzing against a conventional CRV technique when verifying over 480 variations of a sequential FSM circuit, 2) compare Hardware Fuzzing against `RFUZZ` [91] when fuzzing four SiFive TileLink peripherals [143], three RISC-V CPUs [134], and an FFT accelerator [133], and 3) detect four bugs across four commercial IP cores from Google's OpenTitan silicon Root-of-Trust [105].

My main results are summarized as follows. I

- propose deployment of feature-rich software fuzzers as a CDG approach to address inefficiencies in hardware DV (§6.4);
- provide empirically-backed guidance on how to: 1) isolate the DUT portion of HSBs, and 2) minimize overhead of persistent hardware resets, for fuzzing (§6.4.2.1 & §6.6.3);
- design and implement a bus-specific Hardware Fuzzing harness and grammar to facilitate fuzzing all bus-based hardware cores (§6.4.2.3, §6.4.2.4 & §C);
- design, implement, and open-source a HWFP that continuously fuzzes RTL hardware at scale on Google Cloud Platform (GCP) (§6.5);
- demonstrate Hardware Fuzzing provides two orders-of-magnitude reduction in run time and achieves better FSM coverage than current state-of-the-art CRV schemes (§6.6.4);
- demonstrate Hardware Fuzzing achieves 24.76% better HDL line coverage (on average) after 24 hours of fuzzing compared with existing hardware fuzzing approaches, i.e., `RFUZZ` [91] (§6.7.1),
- demonstrate Hardware Fuzzing identify all four RTL bugs in OpenTitan cores faster than alternative approaches (§6.7.2).

6.2 Background

There are two main hardware verification methods: 1) *dynamic* and 2) *formal*. While there have been significant advancements in deploying formal methods in DV workflows [80, 105, 204], dynamic verification remains the gold standard due to its scalability towards complex designs [91]. Therefore, I focus on improving *dynamic* verification by leveraging advancements in the software fuzzing community. Below, I provide a brief overview of the current state-of-the-art in dynamic hardware verification, and software fuzzing.

6.2.1 Dynamic Verification of Hardware

Dynamic verification of hardware typically involves three steps:

1. **test generation**,
2. **hardware simulation**, and
3. **test evaluation**.

First, during *test generation*, a sequence of inputs are crafted to stimulate the DUT. Next, the DUT's behavior—in response to the input sequence—is simulated during *hardware simulation*. Lastly, during *test evaluation*, the DUT's simulation behavior is checked for correctness. These three steps are repeated until all interesting DUT behaviors have been explored. How do I know when I have explored all interesting behaviors? To answer this question, verification engineers measure coverage of both: 1) manually defined functional behaviors (functional coverage) [171] and 2) the HDL implementation of the design (code coverage) [77, 130, 159].

6.2.1.1 Test Generation

To maximize efficiency, DV engineers aim to generate as few test vectors as possible that still close coverage. To achieve this goal, they deploy two main test generation strategies: 1) constrained-random and 2) coverage-directed. The former is typically referred to

holistically as *Constrained Random Verification (CRV)*, and the latter as *Coverage Directed Test Generation (CDG)*. CRV is a partially automated test generation technique where manually-defined input sets are randomly combined into transaction sequences [1, 198]. While better than an *entirely* manual approach, CRV still requires some degree of manual tuning to avoid inefficiencies, since the test generator has no knowledge of test coverage. Regardless, CRV remains a popular dynamic verification technique today, and its principles are implemented in two widely deployed (both commercially and academically) hardware DV frameworks: 1) Accellera's Universal Verification Methodology (UVM) framework (SystemVerilog) [1] and 2) the open-source cocotb (Python) framework [177].

To overcome CRV shortcomings, researchers have proposed CDG [22, 39, 47, 49, 54, 72, 91, 148, 163, 185, 204, 206], or using test coverage feedback to drive future test generation. Unlike CRV, CDG does not randomly piece input sequences together in hopes of exploring new design state. Rather, it *mutates* prior input sequences that explore uncovered regions of the design to iteratively expand the coverage boundary. Unfortunately, due to deployability challenges, e.g., slow coverage tracing and limited applicability to a small set of DUTs, CDG has not seen widespread adoption in practice [91]. In this paper, I recognize that existing software fuzzers provide a solution to many of these deployability challenges, and therefore advocate for verifying hardware using software verification tools. The central challenges in making this possible are adapting software fuzzers to verify hardware, widening the scope of supported designs, and increasing automation of verification.

6.2.1.2 Hardware Simulation

While there are several commercial [28, 107, 157] and open-source [145, 192] hardware simulators, most work in the same general manner, as shown in Fig. 6.2. First, they translate hardware implementations (described in HDL) into a software model, usually in C/C++. Next, they compile the software model and a testbench—either translated from HDL, or implemented in software (C/C++)—and link them with a simulation engine. To-

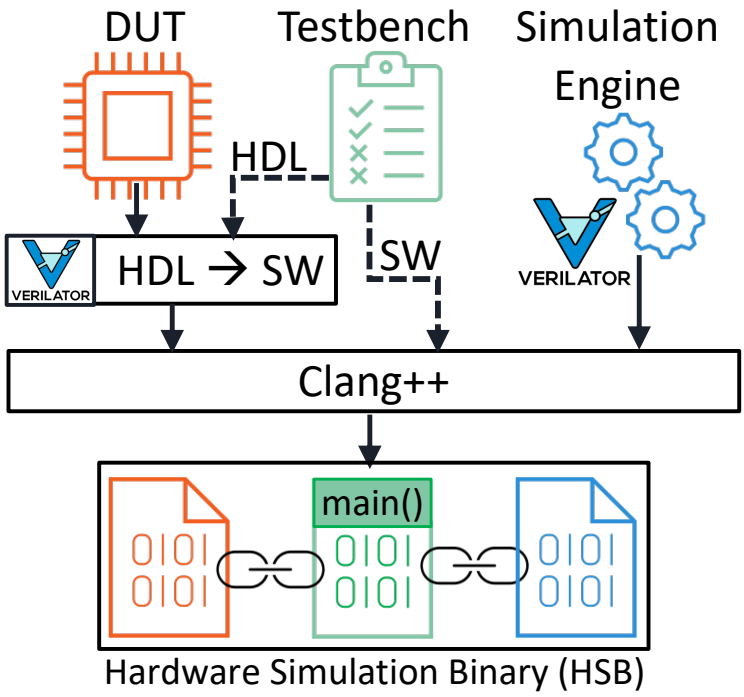

Figure 6.2: **Hardware Simulation Binary (HSB).** To simulate hardware, the DUT's HDL is first translated to a software model, and then compiled/linked with a testbench (written in HDL or software) and simulation engine to form a *Hardware Simulation Binary (HSB)*. Executing this binary with a sequence of test inputs simulates the behavior of the DUT.

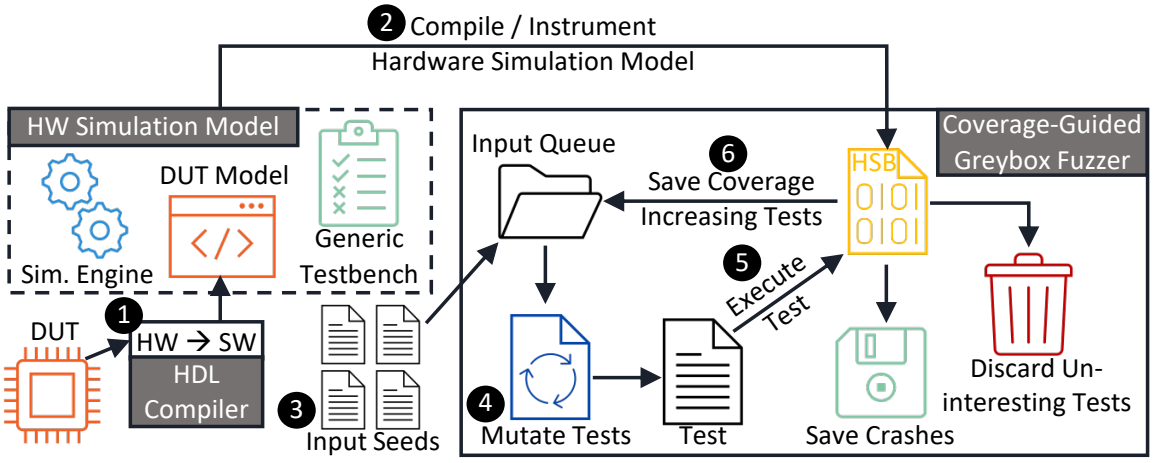

Figure 6.3: **Hardware Fuzzing.** Fuzzing hardware in the software domain involves: translating the hardware DUT to a functionally equivalent software model (1) using a SystemVerilog compiler [145], compiling and instrumenting a Hardware Simulation Binary (HSB) to trace coverage (2), crafting a set of seed input files (3) using our design-agnostic grammar (§ 6.4.2.4), and fuzzing the HSB with a coverage-guided greybox software fuzzer [104, 155, 201] (4–6).

gether, all three components form an *Hardware Simulation Binary (HSB)* (Fig. 6.2) that can be executed to simulate the design. Lastly, the HSB is executed with the inputs from the testbench to capture the design's behavior. Ironically, even though commercial simulators convert the hardware to software, they still **rely on hardware-specific verification tools, likely because software-oriented tools fail to work on hardware models—without the lessons in this paper**. To fuzz hardware in the software domain, I take advantage of the transparency in how an open-source hardware simulator, Verilator [145], generates an HSB. Namely, I *intercept* the software model of the hardware after translation, and instrument/-compile it for coverage-guided fuzzing (Fig. 6.3).

6.2.1.3 Test Evaluation

After simulating a sequence of test inputs, the state of the hardware (both internally and its outputs) are evaluated for correctness. There are two main approaches for verifying design correctness: 1) invariant checking and 2) (gold) model checking. In invariant checking, a set of assertions (e.g., SVAs or software side C/C++ assertions) are used to check properties of the design have not been violated. In model checking, a separate model of the DUT's correct behavior is emulated in software, and compared to the DUT's simulated behavior. I support such features and adopt both invariant violations and golden model mismatches as an analog for software crashes in my hardware fuzzer.

6.2.2 Software Fuzzing

Software fuzzing is an automated testing technique designed to identify security vulnerabilities in software [154]. Thanks to its success, it has seen widespread adoption in both industry [23] and open-source [140] projects. In principle, fuzzing typically involves the following three main steps [116]: 1) **test generation**, 2) **monitoring test execution**, and 3) crash triaging. During test generation, program inputs are synthesized to exercise the target binary. Next, these inputs are fed to the program under test, and its execution is monitored.

Lastly, if a specific test causes a crash, that test is further analyzed to find the root cause. This process is repeated until all, or most, of the target binary has been explored. Below I categorize fuzzers by how they implement the first two steps.

6.2.2.1 Test Generation

Most fuzzers generate test cases in one of two ways, using: 1) a grammar, or 2) mutations. Grammar-based fuzzers [7, 76, 114, 127, 186, 187] use a human-crafted grammar to constrain tests to comply with structural requirements of a specific target application. Alternatively, mutational fuzzers take a correctly formatted test as a seed, and apply mutations to the seed to create new tests. Moreover, mutational fuzzers are tuned to be either: 1) *directed*, or 2) *coverage-guided*. Directed mutational fuzzers [8, 20, 35, 124, 189, 197, 210] favor mutations that explore specific region within the target binary, i.e., prioritizing exploration *location*. Conversely, coverage-guided mutational fuzzers [104, 132, 139, 155, 180, 201] favor mutations that explore as much of the target binary as possible, i.e., prioritizing exploration *completeness*. For this work, I favor the use of mutational, coverage-guided fuzzers, as they are both design-agnostic, and regionally generic.

6.2.2.2 Test Execution Monitoring

Fuzzers monitor test execution using one of three approaches: 1) blackbox, 2) whitebox, or 3) greybox. Fuzzers that only monitor program inputs and outputs are classified as *blackbox* fuzzers [114, 127, 179]. Alternatively, fuzzers that track detailed execution paths through programs with fine-grain program analysis (source code required) and constraint solving are known as *whitebox* fuzzers [24, 33, 38, 50, 65, 149, 189, 199]. Lastly, *greybox* fuzzers [7, 20, 57, 124, 128, 132, 139, 155, 180, 186, 187, 197, 201, 210] offer a trade-off between black- and whitebox fuzzers by deploying lightweight program analysis techniques, such as code-coverage tracing. Since Verilator [145] produces raw C++ source code from RTL hardware, my approach can leverage *any* software fuzzing technique—

white, grey, or blackbox. In my current implementation, I deploy greybox fuzzing, due to its popularity in the software testing community.

6.3 Threat Model

Like Chapter V, I again focus on the *design-time attack* threat model (§2.3.2). My HWFP can fuzz any hardware design produced from any of the first three steps in the hardware design process (Fig. 2.1), provided the design is described in valid HDL, like VHDL or Verilog. I assume the design is not provided in any encrypted format–e.g., an encrypted third party IP block—or if it is, an un-encrypted physical layout of the design is provided in the form of a GDSII file, such that netlist HDL can be reverse engineered from the layout [165]. I *do not* assume the verification engineer utilizing the HWFP has any knowledge regarding the implementation of the design, except for knowledge pertaining to example inputs and (correct) corresponding outputs of the DUT.

6.4 Hardware Fuzzing

To take advantage of advances in software fuzzing for hardware DV, I propose translating hardware designs to software models, and fuzzing the model directly. I call this approach, **Hardware Fuzzing**, and illustrate the process in Fig. 6.3. Below, I first motivate my approach by describing how hardware is already translated to the software domain for simulation, and that software fuzzers provide a solution to a key technical challenge in CDG: scalable coverage tracing. Then, I pose several challenges in adapting software fuzzers to fuzz HSBs (in a design-agnostic fashion), and present solutions to overcome these challenges.

6.4.1 Why Fuzz Hardware like Software?

I observe two key benefits of fuzzing hardware in the software domain. First, hardware is already translated to a software model for simulation purposes (§6.2.1.2). Second, unlike prior CDG approaches [91, 148], I recognize that software fuzzers already provide an efficient solution for tracing coverage. Below I explain how RTL hardware is translated to executable software, and why software fuzzers implicitly maximize hardware coverage by generating tests that maximize coverage of the HSB.

6.4.1.1 Translating HDL to Software

Today, simulating RTL hardware involves translating HDL into a functionally equivalent software (C/C++) model that can be compiled and executed (§6.2.1.2). To accomplish this, most hardware simulators [145, 192] contain an RTL compiler to perform the translation. Therefore, I leverage a popular open-source hardware simulator, Verilator [145], to translate SystemVerilog HDL into a cycle-accurate C++ model for fuzzing.

Like many compilers, Verilator first performs lexical analysis and parsing (of the HDL) with the help of Flex [126] and Bison [166], to generate an Abstract Syntax Tree (AST). Then, it performs a series of passes over the AST to resolve parameters, propagate constants, replace *don't cares* (Xs) with random values, eliminate dead code, unroll loops/generate statements, and perform several other optimizations. Finally, Verilator generates C++ (or SystemC) code representing a cycle-accurate model of the hardware. It creates a C++ class for each Verilog module, and organizes classes according to the original HDL module hierarchy [204].

To interface with the model, Verilator exposes public member variables for each input/output to the top-level module, and a public `eval()` method (to be called in a loop) in the top C++ class. Each input/output member variable is mapped to single/arrayed `bool`, `uint32_t`, or `uint64_t` data types, depending on the width of each signal. Each call to `eval()` updates the model based on the current values assigned to top-level inputs and in-

ternal states variables. Two calls represent a single clock cycle (one call for each rising and falling clock edges).

6.4.1.2 Tracing Hardware Coverage in Software

To efficiently explore a DUT's state space, CDG techniques rely on tracing coverage of past test cases to generate future test cases. There are two main categories of coverage metrics used in hardware verification [77, 130, 159]: 1) *code coverage*, and 2) *functional coverage*. The coarsest, and most used, code coverage metric is *line coverage*. Line coverage measures the percentage of HDL lines that have been exercised during simulation. Alternatively, *functional coverage* measures the percentage of various high-level design functionalities—defined using special HDL constructs like SystemVerilog Coverage Points/Groups—that are exercised during simulation. Regardless of the coverage metric used, tracing HDL coverage during simulation is often slow, since coverage traced in the software (simulation) domain must be mapped back to the hardware domain [77].

In an effort to compute DUT coverage efficiently, and in an HDL-agnostic manner, prior CDG techniques develop custom coverage metrics, e.g., *multiplexer coverage* [91], that can be monitored by instrumenting the RTL directly. However, this approach has two drawbacks. First, the hardware must be simulated on an FPGA (simulating within software is just as slow). Second, the authors provide no indication that their custom coverage metrics actually translate to coverage metrics DV engineers care about.

Rather than make incremental improvements to existing CDG techniques, I recognize that: 1) software fuzzers provide an efficient mechanism—e.g., binary instrumentation—to trace coverage of compiled C++ hardware models (HSBs), and 2) characteristics of how Verilator translates RTL hardware to software makes mapping software coverage to hardware coverage implicit. On the software side, there are three main code coverage metrics of increasing granularity: 1) basic block, 2) basic block edges, and 3) basic block paths [116]. The most popular coverage-guided fuzzers—AFL [201], libFuzzer [104], and

honggfuzz [155]—all trace *edge* coverage. On the hardware side, Verilator conveniently generates straight-line C++ code for both blocking and non-blocking[2] SystemVerilog statements [204], and injects conditional code blocks (basic blocks) for SystemVerilog Assertions and Coverage Points. Therefore, **optimizing test-generation for *edge* coverage of the software model of the hardware during simulation, translates to optimizing for *code*, FSM, and *functional* coverage of the RTL hardware itself.** I demonstrate this artifact in §6.6.4, §6.7.1–6.7.2, and Appendix C.

6.4.2 Driving Hardware with Software Fuzzers

While software fuzzers contain efficient mechanisms for tracing coverage of HSBs—e.g., binary instrumentation—interfacing them with HSBs, in a design-agnostic manner is non-trivial. Below, I highlight several challenges in fuzzing HSBs with software fuzzers, and propose solutions to overcome them.

6.4.2.1 Interfacing Software Fuzzers with HSBs

Naïvely, a DV engineer may interface the HSB directly with a software fuzzer (like [104, 155, 201]) by compiling the HSB source code alongside the testbench harness (Algo. 2) and simulation engine with one of the fuzzer-provided wrappers for Clang. However, they would be ignoring two key differences between typical software applications and HSBs that may degrade fuzzer performance. First, HSBs have other components—a testbench and simulation engine (Fig. 6.2)—that are not part of the DUT. While the DUT is manipulated through the testbench and simulation engine, instrumenting all components HSBs actually degrades fuzzer performance (§6.6.3.1). Additionally, unlike software, the DUT software model must be reset and initialized, prior to processing any inputs. Depending on the size of the DUT, this process can require special configuration of the testbench, i.e., initializing the fuzzer to snapshot the hardware simulation process *after* reset and initialization

[2]Verilator imposes an order on the non-blocking assignments since C++ does not have a semantically equivalent assignment operator [145, 204]. Regardless, this ordering does not effect code coverage.

of the DUT (§6.6.3.2).

6.4.2.2 Interpreting Fuzzer-Generated Tests

For most software, a single input often activates an entire set of state transitions within the program. Consequently, the most popular software fuzzers assume the target binary reads a single dimensional input—e.g., a single image or document—from either a file, `stdin`, or a byte array [104, 155, 201]. As Laeufer *et al.* point out [91], the execution model of hardware is different. In an HSB, a *sequence* of inputs is required to activate state transitions within the DUT. For example, a 4-digit lock (with a keypad) only has a *chance* of unlocking if a sequence of four inputs (test cases) are provided. Fuzzing this lock with single test cases (digits), will fail. Likewise, fuzzing HSBs with software fuzzers that employ a *single-test-case-per-file* model will also fail. Therefore, to stimulate hardware with software fuzzers, I interpret single dimensional fuzzer-generated tests in two dimensions: space and time. I implement this interface in the form of a generic fuzzing harness (testbench)—shown in Algo. 2—that continuously: 1) reads byte-level portions of fuzzer-generated test files, 2) maps these bytes to hardware input ports, and 3) advances the simulation clock by calling the model's `eval()` method twice, until there are no remaining bytes to process. With my fuzzing harness, I transform one-dimensional test inputs, into a two-dimensional *sequence* of inputs.

6.4.2.3 Bus-Centric Harness

While the multi-dimensional fuzzing interface I develop enables fuzzer-generated tests to effect state transitions in hardware, it is not design-agnostic. Specifically, the ports of a hardware model are not iterable (Algo. 2: line 4). A DV engineer would have to create a unique fuzz harness (testbench) for each DUT they verify. To facilitate DUT portability, I take inspiration from how hardware engineers interface IP cores within an SoC [41]. Specifically, I propose fuzzing IP cores at the bus interface using a bus-centric harness.

Algorithm 2: Generic Hardware Fuzzing harness (testbench) that maps one-dimensional fuzzer-generated test files to both spatial and temporal dimensions.

```
   Input: fuzz_test_file.hwf
1  dut ← Vtop();
2  tf ← open(fuzz_test_file.hwf);
3  while tf not empty do
4  |  foreach port ∈ dut.inputs do
5  |  |  tf.read((uint_8t*) port, sizeof(port));
6  |  |  for k ← 1 to 2 do
7  |  |  |  clock ← (clock + 1) % 2;
8  |  |  |  dut.eval();
9  |  |  end
10 |  end
11 end
```

To implement this harness, I could alter my prior harness (Algo. 2) by mapping bytes from fuzzer-generated test files to temporal values for specific signals of a bus-protocol of my choice. However, this would create an exploration barrier since bus-protocols require structured syntax, and most mutational fuzzers lack syntax awareness [200]. In other words, the fuzzer would likely get stuck trying to synthesize a test file, that when mapped to spatio-temporal bus signal values, produces a valid bus-transaction. Instead, I implement a harness that decodes fuzzer-generated test files into sequences of properly structured bus transactions using a bus-centric grammar I describe below. My current bus-centric harness is implemented around the TL-UL bus protocol [70] with a 32-bit data bus, and illustrated in Fig. 6.13.

6.4.2.4 Bus-Centric Grammar

To translate fuzzer-generated test files into valid bus transactions I construct a Hardware Fuzzing grammar. I format my grammar in a compact binary representation to facilitate integration with popular greybox fuzzers that produce similar formats [104, 155, 201]. To match my bus-centric harness, I implement my grammar around the same TL-UL bus protocol [70]. My grammar consists of *Hardware Fuzzing instructions* (Fig. 6.4), that

Figure 6.4: **Hardware Fuzzing Instruction.** A bus-centric harness (testbench) reads binary *Hardware Fuzzing Instructions* from a fuzzer-generated test file, decodes them, and performs TL-UL bus transactions to drive the DUT (Fig.6.13). Our *Hardware Fuzzing Instructions* comprise a grammar (Tbl. 6.1) that aid syntax-blind coverage-guided greybox fuzzers in generating valid bus-transactions to fuzz hardware.

contain: 1) an 8-bit opcode, 2) 32-bit address field, and 3) 32-bit data field. The opcode within each instruction determines the bus transaction the harness performs. I describe the mappings between opcodes and TL-UL bus transactions in Table 6.1.

Note, there are two properties of my grammar that leave room for various harness (testbench) implementations, which I study in Appendix C. First, while I define only three opcodes in my grammar, I represent the opcode with an entire byte, leaving it up to the harness to decide how to map Hardware Fuzzing opcode values to testbench actions. I do this for two reasons: 1) a byte is the smallest addressable unit in most software, facilitating the development of utilities to automate generating compact binary seed files (that comply with my grammar) from high-level markdown languages, and 2) choosing a larger opcode field enables adding more opcodes in the future, should I need to support additional operations in the TileLink bus protocol[70]. Second, of the three opcodes I include, not all require address and data fields. Therefore, it is up to the harness to decide how it should process Hardware Fuzzing instructions. While different implementations may choose to read *fixed* size instruction frames, from my empirical analysis in Appendix C, I decide to implement a harness that processes *variable* size instructions frames, depending on the opcode (Table 6.1).

Table 6.1: Hardware Fuzzing Grammar.

Opcode	Address Required?	Data Required?	Testbench Action
wait	no	no	advance the clock one period
read	yes	no	TL-UL Get (read)
write	yes	yes	TL-UL PutFullData (write)

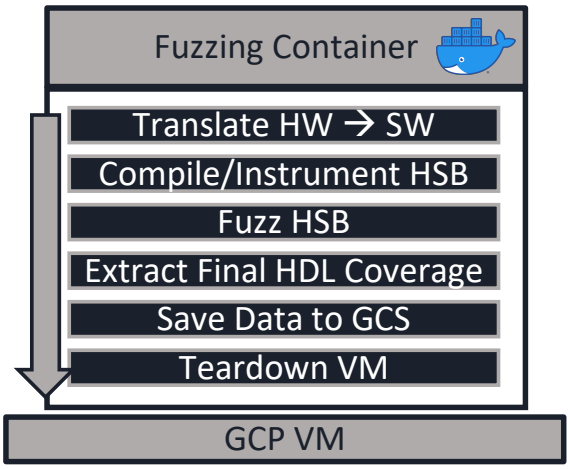

Figure 6.5: **Hardware Fuzzing Pipeline (HWFP).** We design, implement, and open-source a HWFP that is modeled after Google's OSS-Fuzz [140]. Our HWFP enables us to verify RTL hardware at scale using only open-source tools, a rarity in hardware DV.

6.5 Hardware Fuzzing Pipeline

To fuzz hardware at scale I design, implement, and open-source a Hardware Fuzzing Pipeline (HWFP) modeled after Google's OSS-Fuzz (Fig. 6.5). First, my pipeline builds a Docker image (from the Ubuntu 20.04 base image) containing a compiler (LLVM version 12.0.0), RTL simulator (Verilator [145] version 4.0.4), software fuzzer, the target RTL hardware, and a generic fuzzing harness (§6.4.2.3). From the image, a container is instantiated on a GCP VM that:

1. translates the DUT's RTL to a software model with Verilator [145],

2. compiles/instruments the DUT model, and links it with the generic fuzzing harness (§6.4.2.3) and simulation engine to create an HSB (Fig. 6.2),

3. launches the fuzzer for a set period of time, using the `timeout` utility,

4. traces final HDL coverage of fuzzer-generated tests with Verilator [145],

5. saves fuzzing and coverage data to a Google Cloud Storage (GCS) bucket, and lastly

6. tears down the VM.

Note, for benchmarking, all containers are instantiated on their own GCP `n1-standard-2` VM with two vCPUs, 7.5 GB of memory, 50 GB of disk, running Google's Container-Optimized OS. In my current implementation, I use AFL [201] (version 2.57b) as my fuzzer, but my HWFP is designed to be fuzzer-agnostic.

Unlike traditional hardware verification toolchains, my HWFP uses *only* open-source tools, allowing DV engineers to save money on licenses, and spend it on compute. This not only enhances the deployability of my approach, but makes it ideal for adopting alongside existing hardware DV workflows. This is important because rarely are new DV approaches adopted without some overlap with prior (proven) techniques, since mistakes during hardware verification have costly repercussions.

6.6 Feasibility Evaluation

In the first part of my evaluation, I address two technical questions around fuzzing software models of RTL hardware with software fuzzers. First, *how should I interface coverage-guided software fuzzers with HSBs?* Unlike most software, HSBs contain other components—a testbench and simulation engine (Fig. 6.2)—that are *not* the target of testing, yet the fuzzer must learn to manipulate in order to drive the DUT. Second, *how does Hardware Fuzzing compare with traditional dynamic verification methods, i.e., CRV, in terms of time to coverage convergence?* To address this first set of questions, I perform several E2E fuzzing analyses on over 480 digital lock hardware designs with varying state-space complexities.

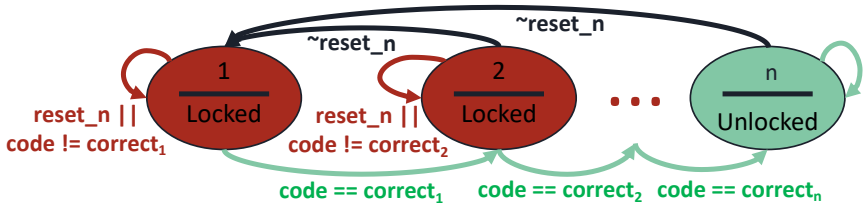

Figure 6.6: **Digital Lock FSM.** We use a configurable digital lock (FSM shown here) to demonstrate: 1) how to interface software fuzzers with hardware simulation binaries, and 2) the advantages of Hardware Fuzzing (vs. traditional CRV). The digital lock FSM can be configured in two dimensions: 1) total number of states and 2) width (in bits) of input codes.

6.6.1 Digital Lock Hardware

In this half of my evaluation, I fuzz various configurations of a digital lock, whose FSM and HDL are shown in Fig. 6.6 and List. VI.1, respectively. I choose to study this design since the complexity of its state space is configurable, and therefore, ideal for stress testing various DV methodologies. Specifically, the complexity is configurable in two dimensions: 1) the total number of states is configurable by tuning the size, N, of the single state register, and 2) the probability of choosing the correct unlocking code sequence is adjustable by altering the size, M, of the comparator/mux that checks input codes against hard-coded (random) values (List. VI.1). I develop a utility in Rust, using the `kaze` crate [161], to auto-generate 480 different lock state machines of various complexities, i.e., different values of N, M, and random correct code sequences.

Listing VI.1: SystemVerilog of Lock with N=$log2(\#states)$ and M-bit secret codes set to random values.

```
module lock(
  input  reset_n ,
  input  clk ,
  input  [M-1:0] code,
  output unlocked
);
logic  [N-1:0] state ;
logic  [M-1:0] correct_codes  [N];

// Secret codes set to random values
for (genvar i = 0; i < N; i++) begin : secret_codes
  assign  correct_codes [i] = <random value>;
```

```
13 end
14
15 assign unlocked = ( state == 'd1) ? 1'b1 : 1'b0;
16
17 always @(posedge clk) begin
18   if (! reset_n ) begin
19     state <= 'd0;
20   end else if (!unlocked && code == correct_codes[ state ]) begin
21     state <= state + 1'b1;
22   end else begin
23     state <= state;
24   end
25 end
26 endmodule
```

6.6.2 Digital Lock HSB Architectures

To study these designs, I construct two HSB architectures (Fig. 6.7) using two hardware DV methodologies: CRV and Hardware Fuzzing. The CRV architecture (Fig. 6.7A) attempts to unlock the lock through a brute-force approach, where random code sequences are driven into the DUT until the *unlocked* state is reached. If the random sequence fails to unlock the lock, the DUT is reset, and a new random sequence is supplied. If the sequence succeeds, an SVA is violated, which terminates the simulation. The random code sequences are *constrained* in the sense that only valid code sequences are driven into the DUT, i.e., 1) each code in the sequence is in the range $[0, 2^M)$ for locks with M-bit code comparators, and 2) sequences contain exactly $2^N - 1$ input codes for locks with 2^N states. The CRV testbench is implemented with the cocotb [177] framework and simulations are run with Verilator [145].

Alternatively, the Hardware Fuzzing HSB (Fig. 6.7B) takes input from a software fuzzer that generates code sequences for the DUT. The fuzzer initializes and checkpoints, a process running the HSB (Fig. 6.2), and repeatedly forks this process and tries various code sequence inputs. If an incorrect code sequence is supplied, the fuzzer forks a new process (equivalent to resetting the DUT) and tries again. If the correct code sequence is provided, an SVA is violated, which the fuzzer registers as a program crash. The difference between

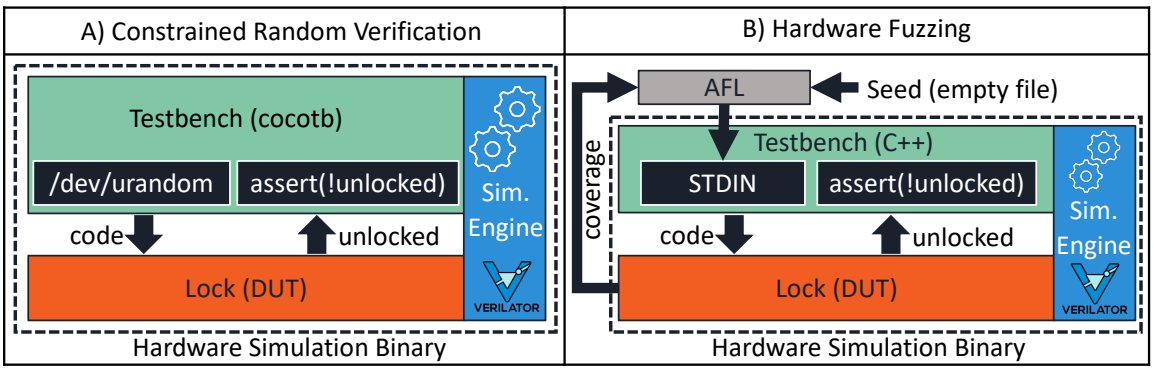

Figure 6.7: **Digital Lock HSB Architectures.** (A) A traditional CRV architecture: random input code sequences are driven into the DUT until the unlocked state is reached. (B) A software fuzzer generates tests to drive the DUT. The fuzzer monitors coverage of the DUT during test execution and uses this information to generate future tests. Both HSBs are configured to terminate execution upon unlocking the lock using an SVA in the testbench that signals the simulation engine (Fig. 6.2) to abort.

CRV and Hardware Fuzzing is that the fuzzer traces coverage during hardware simulation, and will *save* past code sequences that get closer to unlocking the lock. These past sequences are then mutated to generate future sequences. Thus, past inputs are used to craft more *intelligent* inputs in the future. To interface the software fuzzer with the HSB, I:

1. implement a C++ testbench harness from Algo. 2 that reads fuzzer-generated bytes from `stdin` and feeds them directly to the `code` input of the lock.

2. instrument the HSB containing the DUT by compiling it with `afl-clang-fast++`.

6.6.3 Interfacing Software Fuzzers with Hardware

There are two questions that arise when interfacing software fuzzers with HSBs. First, unlike most software applications, software models of hardware are not standalone binaries. They must be combined—typically by either static or dynamic linking—with a testbench and simulation engine to form an HSB (§6.2.1.2). Of these three components—DUT, testbench, and simulation engine—I seek to maximize coverage of *only* the DUT. I do not want

to waste fuzzing cycles on the testbench or simulation engine. Since coverage tracing instrumentation provides an indirect method to coarsely steer the fuzzer towards components of interest [20], it would be considered good practice to instrument just the DUT portion of the HSB. However, while the DUT is ultimately what I want to fuzz, the fuzzer must learn to use the testbench and simulation engine to manipulate the DUT. Therefore, *what components of the HSB should I instrument to maximize fuzzer performance, yet ensure coverage convergence?*

Second, when simulating hardware, the DUT must be reset to a clean state *before* it can start processing inputs. Traditionally, the testbench portion of the HSB performs this reset by asserting the DUT's global reset signal for a set number of clock cycles. Since the fuzzer instantiates, and repeatedly forks the process executing the HSB, this reset process will happen hundreds, or (potentially) thousands of times per second as each test execution is processed. While some software fuzzers [104, 201] enable users to perform initialization operations *before* the program under test is forked—meaning the DUT reset could be performed once, as each forking operation essentially sets the HSB back to a clean state—-this may not always the case. Moreover, it complicates fuzzer–HSB integration, which contradicts the whole premise of my approach, i.e., low-overhead, design-agnostic CDG. Therefore, I ask: *is this fuzzing initialization feature required to fuzz HSBs?*

6.6.3.1 Instrumenting HSBs for Fuzzing

To determine the components of the HSB I should instrument, I measure the fuzzing run times to achieve approximate full FSM coverage[3] of several lock designs, i.e., the time it takes the fuzzer to generate a sequence of input codes that *unlocks each lock*. I measure this by modifying the fuzzer to terminate upon detecting the first crash, which I produce using a single SVA that monitors the condition of the *unlocked* signal (List. VI.1). Specifically, using lock designs with 16, 32, and 64 states, and input codes widths of four bits, I construct

[3] I use the term *approximate* when referring to *full FSM coverage*, since I are not excising the lock's reset state transitions (Fig. 6.6) in these experiments.

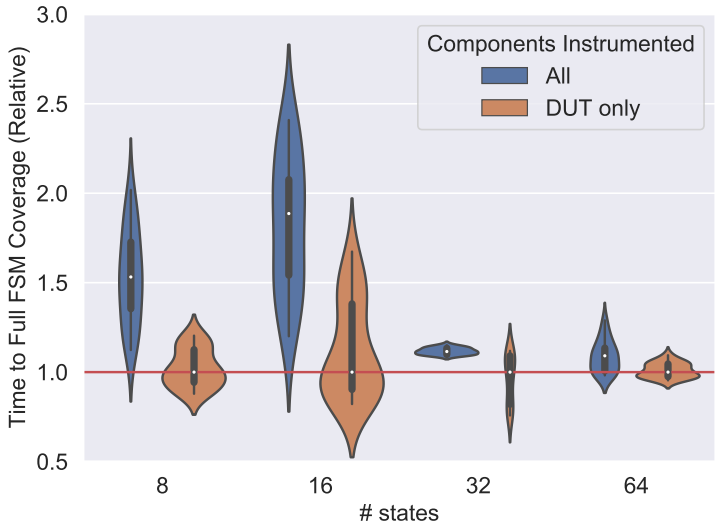

Figure 6.8: **Instrumentation Level vs. Coverage Convergence Rate.** Distribution of fuzzer run times required to *unlock* various sized digital locks (code widths are fixed at four bits), i.e., achieve $\approx$ full FSM coverage. For each HSB, we vary the components we instrument for coverage tracing. Run times are normalized to the median DUT-only instrumentation level (orange) across each lock size (red line). While the fuzzer uses the testbench and simulation engine to manipulate the DUT, instrumenting only the DUT *does not* hinder the coverage convergence rate of the fuzzer. Rather, it improves it when DUT sizes are small, compared to the simulation engine and testbench (Fig. 6.9).

HSBs following the architecture shown in Fig. 6.7B. For each HSB, I vary the components I instrument by using different compiler settings for each component. First, I (naïvely) instrument **all** components, then only the **DUT**. Next, I fuzz each HSB 50 times, seeding the fuzzer with an empty file in each experiment.

I plot the distribution of fuzzing run times in Fig. 6.8. Since fuzzing is an inherently random process, I plot only the middle third of run times across all instrumentation levels and lock sizes. Moreover, all run times are normalized to the median DUT-only instrumentation run times (orange) across each lock size. In addition to plotting fuzzing run times, I plot the number of basic blocks within each component of the HSB in Fig. 6.9. Across all lock sizes, I observe that only instrumenting the DUT does not handicap the fuzzer, but rather *improves the rate of coverage convergence*! In fact, I perform a Mann-Whitney U test, with a 0.05 significance level, and find all the run-time improvements to be statistically significant. Moreover, I observe that even though the run-time improvements are less significant as the DUT size increases compared to the simulation engine and testbench (Fig. 6.9), instrumenting only the DUT never handicaps the fuzzer performance.

Key Insight: Instrumenting only the DUT portion of the HSB does not impair the fuzzer's ability to drive the DUT, rather, it improves fuzzing speed.

6.6.3.2 Hardware Resets vs. Fuzzer Performance

To determine if DUT resets present a performance bottleneck, I measure the degradation in fuzzing performance due to the repeated simulation of DUT resets. I take advantage of a unique feature of a popular greybox fuzzer [201] that enables configuring the exact location of initializing the *fork server*.[4] This enables the fuzzer to perform any program-specific initialization operations *once*, prior to forking children processes to fuzz. Using this feature, I repeat the same fuzzing run time analysis performed in §6.6.3.1, except I

[4]By default, AFL [201] instantiates a process from the binary under test, pauses it, and repeatedly forks it to create identical processes to feed test inputs to. The component of AFL that performs process forking is known as the *fork server*.

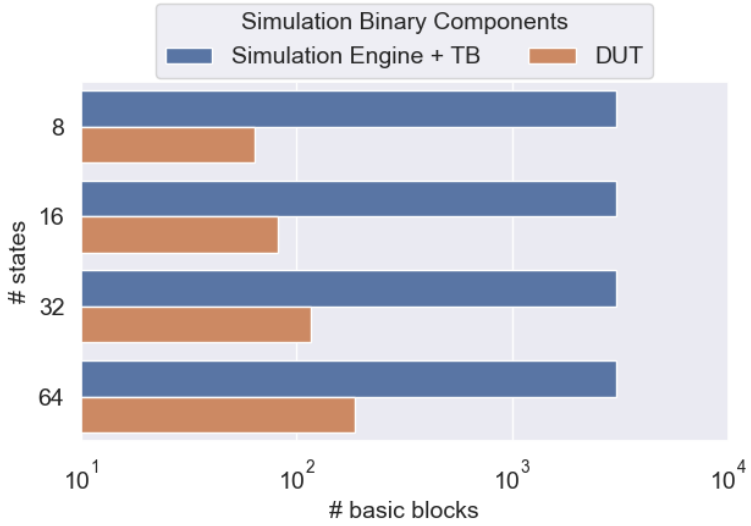

Figure 6.9: **Basic Blocks per Simulation Binary Component.** We break down the number of basic blocks that comprise the three components within HSBs of different size locks (Fig. 6.6 & List. VI.1), generated by Verilator [145]: simulation engine and testbench (TB), and DUT. As locks increase in size, defined by the number of FSM states (code widths are fixed to 4 bits), so do the number of basic blocks in their software model.

instrument all simulation binary components, and compare two variations of the digital lock HSB shown in Fig. 6.7B. In one testbench, I use the default fork server initialization location: at the start of `main()`. In the other testbench, I initialize the fork server *after* the point where the DUT has been reset.

Fig. 6.10 shows my results. Again, I drop outliers by plotting only the middle third of run times across all lock sizes and fork server initialization points. Additionally, I normalize all run times to the median "after DUT reset" run times (orange) across each lock size. From these results, I apply the Mann-Whitney U test (with 0.05 significance level) between run times. This time, only locks with 8 and 16 states yield p-values less than 0.05. This indicates the overhead of continuously resetting the DUT during fuzzing diminishes as the DUT increases in complexity. Additionally, I note that even the largest digital locks I study (64 states), are smaller than the smallest OpenTitan core, the RISC-V Timer, in terms of number of basic blocks in the software model (Fig. 6.9 & Table 6.2).

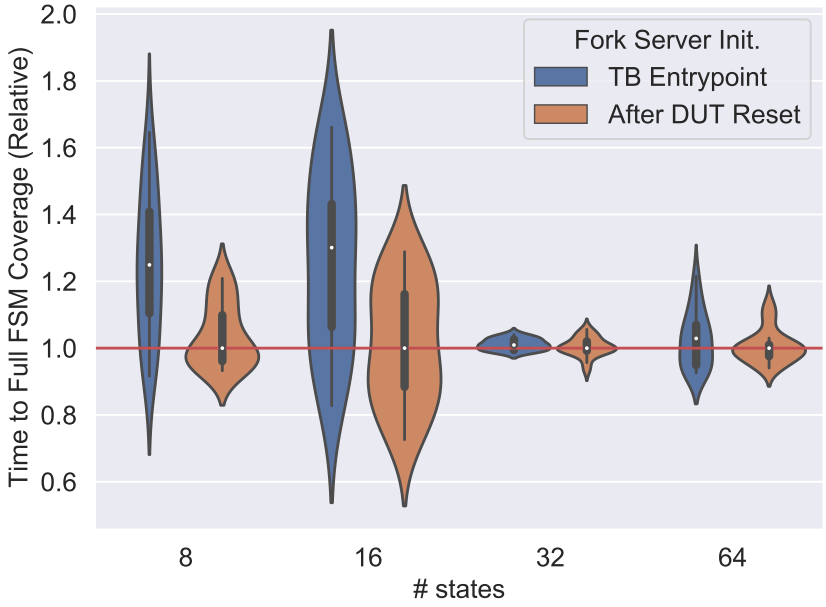

Figure 6.10: **Hardware Resets vs. Fuzzer Performance.** Fuzzing run times across across digital locks (similar to Fig. 6.8) with different fork server initialization locations in the testbench to eliminate overhead due to the repeated simulation of hardware DUT resets. DUT resets are only a fuzzing bottleneck when DUTs are small, reducing fuzzer–HSB integration complexity.

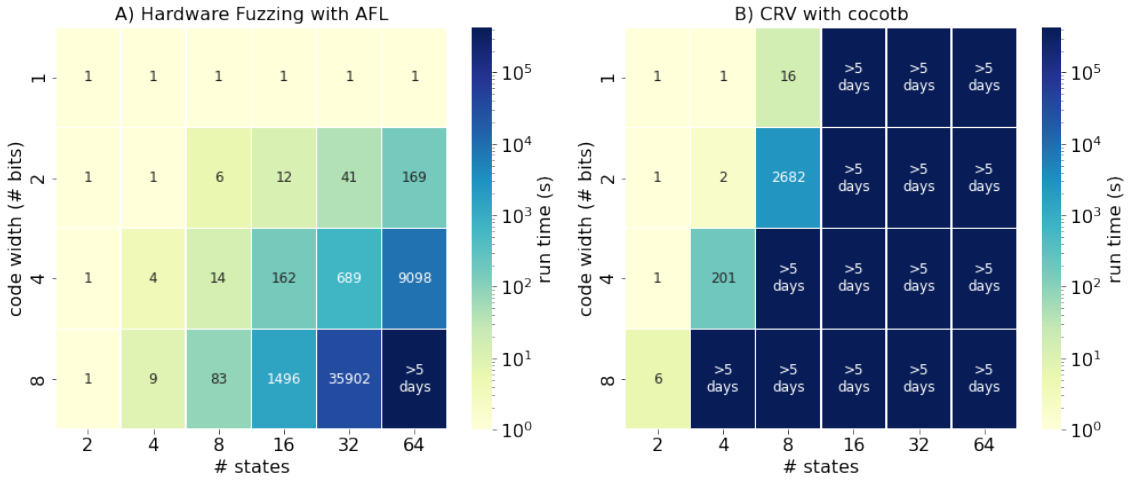

Figure 6.11: **Hardware Fuzzing vs. CRV.** Run times for both Hardware Fuzzing (A) and CRV (B) to achieve ≈ full FSM coverage of various digital lock (Fig. 6.6) designs—i.e., time to unlock the lock—using the testbench architectures shown in Fig. 6.7. Run times are averaged across 20 trials for each lock design—defined by a (# states, code width) pair—and DV method combination. Across these designs, Hardware Fuzzing achieves full FSM coverage faster than traditional CRV approaches, by over two orders of magnitude.

Key Insight: Overhead from simulating hardware resets while fuzzing is minimal, especially in large designs, further reducing fuzzer–HSB integration efforts.

6.6.4 Hardware Fuzzing vs. CRV

Using the techniques I learned from above, I perform a run-time comparison analysis between Hardware Fuzzing and CRV,[5] the current state-of-the-art hardware dynamic verification technique. I perform these experiments using digital locks of various complexities, from 2 to 64 states, and code widths of 1 to 8 bits. The two HSB architectures I compare are shown in Fig. 6.7, and discussed in §6.6.2. Note, the fuzzer was again seeded with an empty file to align its starting state with the CRV tests.

Similar to my instrumentation and reset experiments (§6.6.3) I measure the fuzzing *run times* required to achieve $\approx$ full FSM coverage of each lock design, i.e., the time to *unlock each lock*. I illustrate these run times in heatmaps shown in Fig. 6.11. I perform 20 trials for each experiment and average these run times in each square of a heatmap. While the difference between the two approaches is indistinguishable for extremely small designs, the advantages of Hardware Fuzzing become apparent as designs increase in complexity. For medium to larger lock designs, Hardware Fuzzing achieves full FSM coverage faster than CRV by over two orders-of-magnitude, even when the fuzzer is seeded with an empty file. Moreover, many CRV experiments were terminated early (after running for five days) to save money on GCP instances.

Key Insight: Hardware Fuzzing is a low-cost, low-overhead CDG approach for hardware DV.

[5] CRV is widely deployed in any DV testbenches built around the cocotb [177] or UVM [1] frameworks, e.g., all OpenTitan [105] IP core testbenches.

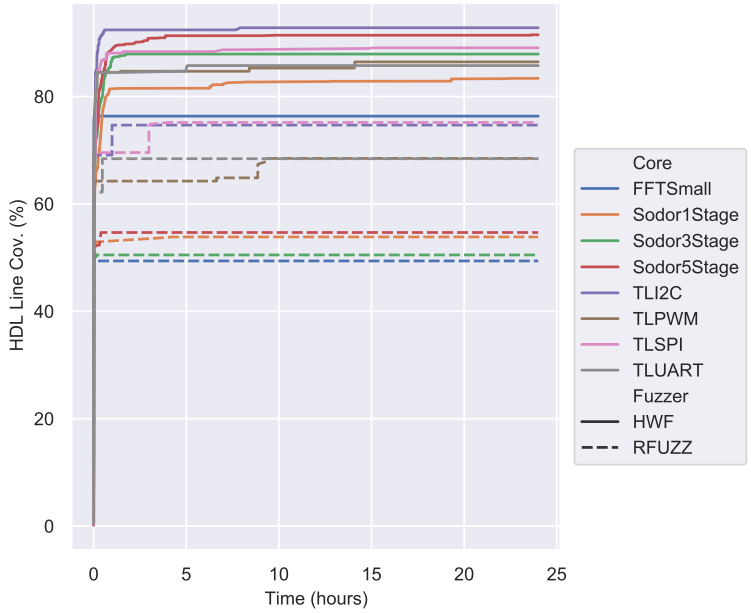

Figure 6.12: **Hardware Fuzzing vs. RFUZZ.** Fuzzing eight different hardware designs, including an FFT accelerator, RISC-V CPUs, and TileLink communication peripherals, with my Hardware Fuzzing approach vs. RFUZZ [91] (Fig. 6.1), yields 24.76% better HDL coverage (on average) after 24 hours, across all cores.

6.7 Practicality Evaluation

In the second part of my evaluation, I address two remaining questions. First, *how does Hardware Fuzzing compare with prior RTL fuzzing schemes, e.g., RFUZZ [91], in terms of HDL code coverage?* While Laeufer *et al.* were the first to demonstrate fuzzing RTL with `RFUZZ` [91], I argue for an entirely different approach (Fig. 6.1), fuzzing software models of RTL hardware, rather than the RTL hardware itself. Lastly, *how does Hardware Fuzzing perform in practice commercial-grade hardware IP?* To address these questions, I perform E2E fuzzing analyses on several open-source hardware designs, including four commercial-grade cores from Google's OpenTitan [105] SoC, four SiFive TileLink peripherals, three RISC-V CPUs, and an FFT accelerator.

6.7.1 Hardware Fuzzing vs. RFUZZ

Unlike my approach, RFUZZ instruments RTL hardware directly by injecting coverage-tracing hardware into the RTL when it is compiled from a high-level HDL, like FIRRTL, to Verilog. As a result, RFUZZ is only compatible with hardware designs described using high-level HDLs, like Chisel [11] or FIRRTL [98]. Unfortunately, most industry hardware designs are still written in (System)Verilog. Moreover, RFUZZ does not exploit any bus-specific harnesses, rather, it requires design-specific harnesses that are fed fuzzer-generated bit-vectors to hardware input ports, as described in Algo. 2 and demonstrated in the fuzzing harness built for the digital lock in Fig. 6.7b.

To demonstrate the benefits of my approach vs. RFUZZ, I compare the HDL line coverage achieved by both approaches over the course of fuzzing eight different hardware designs for 24 hours. Specifically, I fuzz the same eight hardware designs in the original RFUZZ [91], including the I2C, SPI, PWM, and UART SiFive TileLink IP blocks [143], three RISC-V Sodor CPUs [134], and an FFT accelerator [133]. For each core, I use the same RFUZZ-generated test harness across both approaches, but use different fuzzing mechanisms, as highlighted in Fig. 6.1. Specifically, RFUZZ uses a custom fuzzer that directly measures RTL coverage using Verilog-level instrumentation, while my (Hardware Fuzzing) approach uses a software fuzzer (i.e., AFL) that measures RTL coverage using HSB-level instrumentation. For each core, I perform 10 trials with both fuzzing techniques, using empty seed files, and compare the *best case* results (i.e., highest coverage) using RFUZZ, with the *worst case* results (i.e., lowest coverage) using my Hardware Fuzzing approach. In Fig. 6.12, I plot my results. After 24 hours of fuzzing, across all cores, the average HDL line coverage improvement using my Hardware Fuzzing approach over RFUZZ was 24.76%, while the minimum and maximum improvements I 13.90% and 37.41%, respectively. Lastly, I apply the Mann-Whitney U test (with 0.05 significance level) between all fuzzing trials across all cores, and observe p-values less than 0.05. This further confirms my approach yields better coverage than RFUZZ.

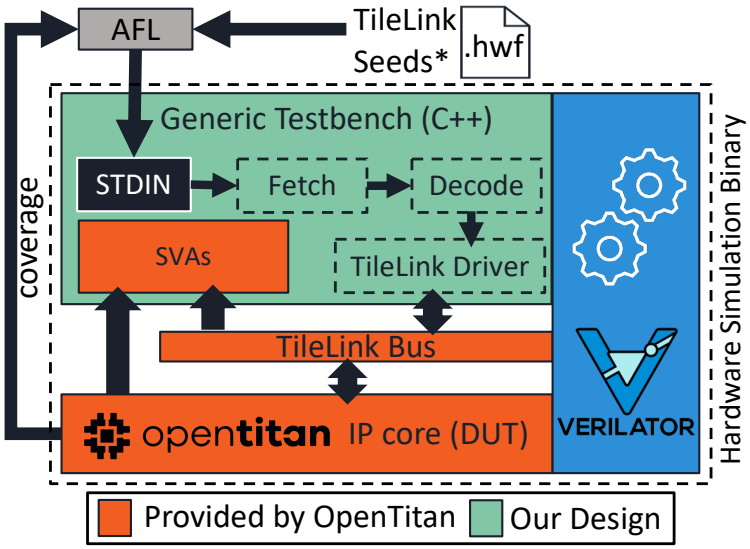

Figure 6.13: **OpenTitan HSB Architecture.** A software fuzzer learns to generate fuzzing *instructions* (Fig. 6.4)—from `.hwf` seed files—based on a hardware fuzzing grammar (§6.4.2.4). It pipes these instructions to `stdin` where a generic C++ fuzzing harness fetches/decodes them, and performs the corresponding TileLink bus operations to drive the DUT. SVAs are evaluated during execution of the HSB, and produce a program crash (if violated), that is caught and reported by the software fuzzer.

Key Insight: Fuzzing hardware in the software domain yields better HDL coverage than prior techniques, i.e. RFUZZ [91].

6.7.2 Fuzzing OpenTitan IP

To address the last question—*How does Hardware Fuzzing perform in practice on commercial-grade hardware?*—I fuzz four IP blocks from Google's OpenTitan silicon root-of-trust SoC[105], including the: AES, HMAC, KMAC, and RISC-V Timer cores. While each core performs different functions,[6] they all conform to the OpenTitan *Comportability Specification* [41], implying **they are all controlled via reads and writes to memory-mapped registers over a TL-UL bus**. By adhering to a uniform bus protocol, I are able to re-use a generic fuzzing harness (Fig. 6.13), facilitating the deployability of my

[6]For more information on the functionalities of each IP block, see Appendix B.

Table 6.2: OpenTitan IP Core Complexity in HW and SW Domains.

IP Core	HW LOC	SW LOC	# Basic Blocks*	# SVAs†
AES	4,562	38,036	3,414	53
HMAC	2,695	18,005	1,764	30
KMAC	4,585	119,297	6,996	44
RV Timer	677	3,111	290	8

* Number of basic blocks in compiled software model with O3 optimization.
† Number of SystemVerilog Assertions included in IP HDL at time of writing.

approach. Below, I highlight the functionality of each IP core. Additionally, in Table 6.2, I report the complexity of each IP core in both the hardware and software domains, in terms of Lines of Code (LOC), number of basic blocks, and number of SVAs provided in each core's HDL. Software models of each hardware design are produced using Verilator, as I describe in §6.4.1.1.

6.7.2.1 Fuzzing OpenTitan IP with Empty Seeds

Unlike most software applications that are fuzzed [140], I observe that software models of hardware are quite small (Table 6.2). So, like the `RFUZZ` experiments, I decided to experiment fuzzing each OpenTitan core using a single empty seed file as starting input, this time for only one hour. I plot the results of this experiment in Fig. 6.14. After only one hour of fuzzing with no proper starting seeds, I achieve over 88% HDL line coverage across three of the four OpenTitan IP cores I study, and over 65% coverage of the remaining design.

6.7.2.2 Fuzzing for Bugs in OpenTitan IP

While coverage is an important metric, the ultimate goal of fuzzing hardware is to automatically uncover bugs, before they percolate into fabricated silicon. Therefore, in my final evaluation, I demonstrate the effectiveness of Hardware Fuzzing at finding four RTL bugs, one in each OpenTitan IP block I study. Specifically, in the AES, HMAC, and

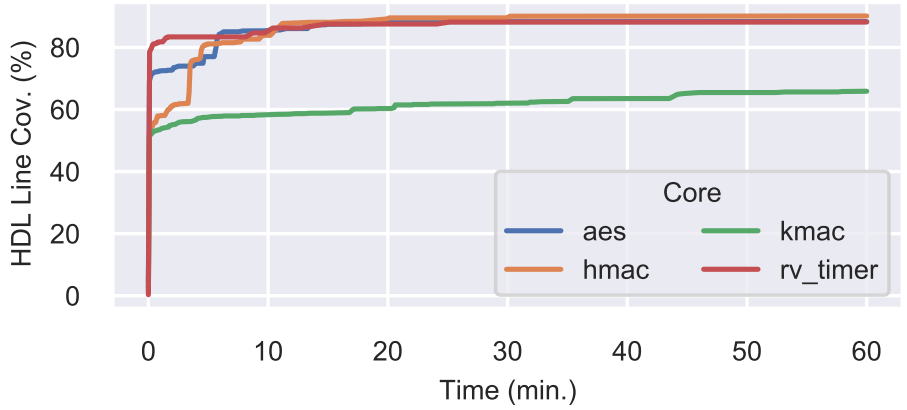

Figure 6.14: **Coverage vs. Time Fuzzing with Empty Seeds.** Fuzzing four OpenTitan [105] IP cores for one hour, seeding the fuzzer with an empty file in each case, yields over 88% HDL line coverage in three out of four designs.

Table 6.3: Hardware Fuzzing RTL Bug Discovery Times.

Core	Bug Type	Time-to-Bug-Discovery (s)
AES	FSM	27
HMAC	Padding	30
KMAC	FSM	35548
RV Timer	Comparison	4.4

RV Timer cores, I implant artificial FSM, padding, and comparison bugs, respectively, and craft corresponding SVAs to produce HSB crashes upon encountering incorrect hardware behaviors. Additionally, for the KMAC core, I craft an SVA to detect an FSM bug that was reported on the OpenTitan public GitHub (Issue #6408) by OpenTitan DV engineers.[7] In Table 6.3, I plot the time it took my HWFP to detect each bug when seeded with a set of inputs that simply resets and initializes each DUT to perform its prescribed tasks. Namely, for the AES core, my seed configures the device to operate in CTR mode. For the HMAC core, my seed configures the device to perform SHA256 hashes. For the RV Timer core, my seed arms the timer. Lastly, for the KMAC core, my seed configures the device to perform KMAC operations in cSHAKE hashing mode. Across each core I study, I are able to detect all implanted bugs in less than 10 hours, with initialization seeds that are orders-of-magnitude less complex than conventional dynamic verification testbenches.

[7] https://github.com/lowRISC/opentitan/issues/6408

Key Insight: Hardware Fuzzing detects bugs in commercial-grade hardware IP.

6.8 Discussion

6.8.1 Detecting Bugs During Fuzzing

The focus of Hardware Fuzzing is to provide a scalable yet flexible solution for integrating CDG with hardware simulation. However, *test generation* and *hardware simulation* comprise only two-thirds of the hardware verification process (§6.2.1). The final, and arguably most important, step is detecting incorrect hardware behavior, i.e., *test evaluation* in §6.2.1.3. For this there are two approaches: 1) invariant checking and 2) (gold) model checking. In both cases, I trigger HSB crashes upon detecting incorrect hardware behavior, which software fuzzers log. For invariant checks, I use SVAs that send the HSB process the `SIGABRT` signal upon assertion violation. Likewise, for gold model checking testbenches any mismatches between models results in a `SIGABRT`.

6.8.2 Additional Bus Protocols

To provide a design-agnostic interface to fuzz RTL hardware, I develop a design-agnostic testbench harness (Fig. 6.13). My harness decodes fuzzer-generated tests using a bus-specific grammar (§6.4.2.4), and produces corresponding TL-UL bus transactions that drive a DUT. In my current implementation, my generic testbench harness conforms to the TL-UL bus protocol [70]. As a result, I can fuzz any IP core that speaks the same bus protocol (e.g., all OpenTitan cores [105]). To fuzz cores that speak other bus protocols (e.g., Wishbone, AMBA, Avalon, etc.), users can simply write a new harness for the bus they wish to support.

6.8.3 Hardware without a Bus Interface

For hardware cores that perform I/O over a generic set of ports that do not conform to any bus protocol, I provide a generic testbench harness that maps fuzzer-generated input files across spatial and temporal domains by interpreting each fuzzer-generated file as a sequence of DUT inputs (Algo. 2). I demonstrate this Hardware Fuzzing configuration when fuzzing various digital locks (Fig. 6.7B). However, if inputs require any structural dependencies, I advise developing a grammar and corresponding testbench—similar to my bus-specific grammar (§6.4.2.4)—to aid the fuzzer in generating valid test cases. Designers can use the lessons in this paper to guide their core-specific grammar designs.

6.8.4 Limitations

While Hardware Fuzzing is both efficient and design-agnostic, there are some limitations. First, unlike software there is no notion of a *hardware sanitizer*, that can add safeguards against generic classes of hardware bugs for the fuzzer to sniff out. While I envision hardware sanitizers being a future active research area, for now, DV engineers must create invariants or gold models to check design behavior against for the fuzzer to find crashing inputs. Second, there is notion of analog behavior in RTL hardware, let along in translated software models. In its current implementation, Hardware Fuzzing is not effective against detecting side-channel vulnerabilities that rely on information transmission/leakage through analog domains.

6.9 Related Work

There are two categories of prior CDG approaches: 1) design-agnostic and 2) design-specific.

6.9.1 Design-Agnostic

Laeufer *et al.*'s RFUZZ [91] is the most relevant prior work, which attempts to build a full-fledged design-agnostic RTL fuzzer. To achieve their goal, they propose a new RTL coverage metric—*mux toggle coverage*—that measures if the control signal to a 2:1 multiplexer expresses both states (0 and 1). Unlike Hardware Fuzzing, they instrument the RTL directly, by injecting additional HDL into the design, and develop their own custom RTL fuzzer (Fig. 6.1). Unfortunately, this has three drawbacks. First, RFUZZ is only compatible with hardware written in high-level HDLs, like Chisel [11] or FIRRTL [98], that can instrumented when compiled to Verilog. Second, RFUZZ requires some designs be modified to have reset times on the order of one to two clock cycles. Third, it is unclear how their *mux toggle coverage* maps to other RTL coverage metrics that DV engineers also care about, e.g., FSM and functional coverage [77, 159]. Gent *et al.* [49] also propose an automatic test pattern generator based on custom coverage metrics, for which they too instrument the RTL directly to trace. Unfortunately, like RFUZZ, the deployability and scalability of their approach remains in question, given their coverage tracing method.

6.9.2 Design-Specific

Unlike the *design-agnostic* approaches, several researchers propose CDG techniques exclusively for processors. Zhang *et al.* [204] propose Coppelia, a tool that uses a custom symbolic execution engine (built on top of KLEE [24]) on software models of the RTL. Coppelia's goal is to target specific security-critical properties of processors; Hardware Fuzzing enables combining such static methods with fuzzing (i.e., concolic execution [149]) for free, overcoming the limits of symbolic execution alone. Hur *et al.* [67] propose DIFUZZRTL that combines RFUZZ with golden model checking to find bugs in CPUs. However, Hardware Fuzzing produces better coverage than RFUZZ (§6.7.1), and can be combined with invariant *or* with golden model checking to detect bugs. Lastly, two other processor-specific CDG approaches are Squillero's *MicroGP* [148] and Bose *et al.*'s [22]

that use a genetic algorithms to generate random assembly programs that maximize RTL code coverage of a processor. Unlike Hardware Fuzzing, these approaches require custom DUT-specific grammars to build assembly programs from.

6.10 Conclusion

Hardware Fuzzing is an effective solution to CDG for hardware DV. Unlike prior work, I take advantage of feature-rich software testing methodologies and tools, to solve a long-standing problem in hardware DV. To make my approach attractive to DV practitioners, I solve several key deployability challenges, including developing generic interfaces (grammar & testbench) to fuzz RTL in a design-agnostic manner. Using my generic grammar and testbench, I show that my Hardware Fuzzing approach can achieve over 88% HDL code coverage of three out of four commercial-grade hardware designs I study in only one hour, with no knowledge of the DUT design or implementation. Moreover, I demonstrate that approach can also detect various implanted bugs in the same designs, in less than 10 hours. Finally, compared to standard dynamic verification practices and prior RTL fuzzing techniques, with Hardware Fuzzing, I achieve over two orders-of-magnitude and 13.90% coverage convergence improvements, respectively.

6.11 Citation

Work from this chapter was partially completed while interning at Google, and is co-authored by Kang G. Shin, Alex Chernyakovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. This work can be cited as [172].

CHAPTER VII

Conclusion & Future Directions

7.1 Conclusion

The ultimate goal of this research is to make security a first class optimization objective—alongside power, performance, and area—throughout the hardware development life cycle. In this dissertation, I developed four techniques to aid in the design and verification of secure hardware. Specifically, I addressed two security threats related to: 1) fabricating silicon at untrusted foundries, and 2) integrating untrusted third-party IP into larger (trusted) designs. Unlike software, hardware cannot be patched once deployed. Any flaws or security vulnerabilities in hardware have detrimental financial repercussions to companies that experience them. As hardware becomes increasingly sophisticated and application-specific, it is more important than ever to root out hardware flaws *before* fabrication, and to *prevent* malicious tampering of designs at untrusted foundries.

To summarize my contributions to the field, I take a bottom-up approach to securing IC hardware: starting at the layout-level, before moving to the behavioral (RTL) level. First, I presented a framework to compute metrics that quantify the overall security of an IC layout to fabrication-time modification. The goal of this framework is to provide an optimization feedback mechanism to Electronic Design Automation (EDA) tools to enhance the security of IC layouts. Using this measurement framework, I developed the first routing-based preventative defense against foundry-side attacks, called T-TER (Chapter IV). Unlike prior

defenses, T-TER does not require hard layout constraints that make deployment intractable in real-world designs.

At the behavioral (RTL) level, I developed verification solutions to vet untrusted third-party IP for hardware Trojans. In Chapter V, I presented the first dynamic verification technique—*Bomberman*—capable of detecting TTTs with zero false negatives. Lastly, using Bomberman's insights, I developed a design-agnostic technique to fuzz hardware like software to automatically identify specific RTL hardware vulnerabilities, using feature-rich coverage-guided software fuzzers (Chapter VI). Hardware fuzzing is an exciting research domain as it brings together several recent advancements in coverage-guided software testing to solve stagnant hardware verification problems.

7.2 Future Directions

While my research has made strides towards securing hardware, it has only scratched the surface in what I believe to be the future of computer security research. Specifically, my work has brought to light several new challenges that I summarize here.

7.2.1 Security as an Optimization Objective during IC Layout

In Chapter III, I presented a framework for computing security metrics of an IC layout. While these metrics provide IC layout engineers with concrete insights into how their designs might be vulnerable to fabrication-time attacks, they do not provide an automated mechanism to address any issues that may come to light. Ultimately, the goal of this framework is to be tightly integrated into PaR EDA tools, such that the tools themselves can optimize IC layouts for security. I envision future work performing this integration to develop an E2E solution for hardening IC layouts against fabrication-time modifications.

7.2.2 Directed Fuzzing for Trojan Detection

The Bomberman toolchain presented in Chapter V demonstrates the effectiveness of Trojan-specific verification at eliminating the possibility of false negatives when vetting untrusted third-party IP for Trojans. Unfortunately, as my results show, the false positive rate of this technique is largely dependent on the verification test coverage of TTT invariants—e.g., repeated or exhausted counter values—Bomberman monitors. While Bomberman itself provides insight into how to improve the said test coverage, it requires manual effort on behalf of the design verification engineer. Moreover, it assumes design verification engineer's have intimate knowledge of the implementation of the DUT, which is not the case when vetting third-party IP for Trojans. To address this issue, I foresee future work deploying security-critical invariant monitors, like [59], with hardware fuzzing (Chapter VI), to automatically optimize Trojan-specific detection schemes like Bomberman.

7.2.3 Fuzzing Hardware with Sparse Memories

In Chapter VI, I demonstrated the advantage of deploying software fuzzing tools to the hardware verification domain. Using this technique, I fuzzed several cores from the OpenTitan root-of-trust SoC, including the AES, HMAC, KMAC, and timer cores. One thing all of these cores have in common is they are all *loosely-coupled accelerators*. In other words, unlike a processor, they do *not* interface with sparse memories. Rather, they accept control- and data-path inputs directly from a bus interface, process these inputs, and produce outputs over the same interface. To tailor my approach to such designs, I create a bus-specific grammar and harness to interpret fuzzer-generated inputs as a sequence of bus transactions to drive these DUTs.

Now, consider the scenario where the DUT is a CPU that processes control- and data-path inputs from a large, sparse memory. To adapt software fuzzers to such a DUT, I envision creating an ISA-specific grammar and harness to interpret fuzzer-generated inputs as a valid executable program. However, assuming memory is initialized to a clean

state prior to loading a fuzzer-generated program, most program memory accesses would go to empty memory regions, resulting in fewer interesting state transitions. To increase the probability of the fuzzer exploring interesting hardware states, we need to develop a CPU-specific test mutator that constrains program memory accesses to small, yet random, memory regions.

7.2.4 Hardware Sanitizers

Another key challenge in my hardware fuzzing work was developing a mechanism to signal to the software fuzzer that a hardware bug was found. Traditionally, when fuzzing software, this is accomplished using *software sanitizers* that instrument the program under test such that it crashes when common abnormal behavior is observed, such as a buffer overflow, use-after-return, or memory leak error. Unfortunately, there is no such construct in the hardware domain, as hardware flaws have not been well categorized. To detect incorrect hardware behavior during simulations, hardware verification engineers use golden models or invariant (e.g., SystemVerilog Assertions) checks. In my hardware fuzzing work, I translated these checks to software-level assertions so the fuzzer could detect incorrect hardware behavior. Unfortunately, defining golden models or invariant checks is tedious and requires DUT implementation knowledge. My vision going forward is to develop a *hardware sanitizer* to automate instrumenting software models of RTL hardware with invariant checks that detect the most common RTL bugs found in open-source hardware, e.g., [59].

APPENDICES

APPENDIX A

Route Distances of OR1200 Layouts

A target density of 50–90% was held across each layout, while target clock frequency and max transition time parameters were varied from 100 MHz to 1000 MHz and 100 ps to 300 ps respectively. Each heatmap in Figures A.1–A.3 is intended to be read column-wise, where each column is a histogram. The color intensity within a heatmap column indicates the percentage of (critical-net, trigger-space) pairs, within that column, that are within a range of distance away. The y-axis reports the distance in terms of standard deviations from the overall mean net-length in each design. The x-axis reports the trigger space sizes in number of contiguous placement sites. Designs with smaller trigger-spaces and long route distances are more resistant to fabrication-time attacks. Namely, a heatmap column that is completely dark indicates no (critical-net, trigger-space) pairs, or attack points, and a column that is completely dark except for the top-most cell is the second most secure.

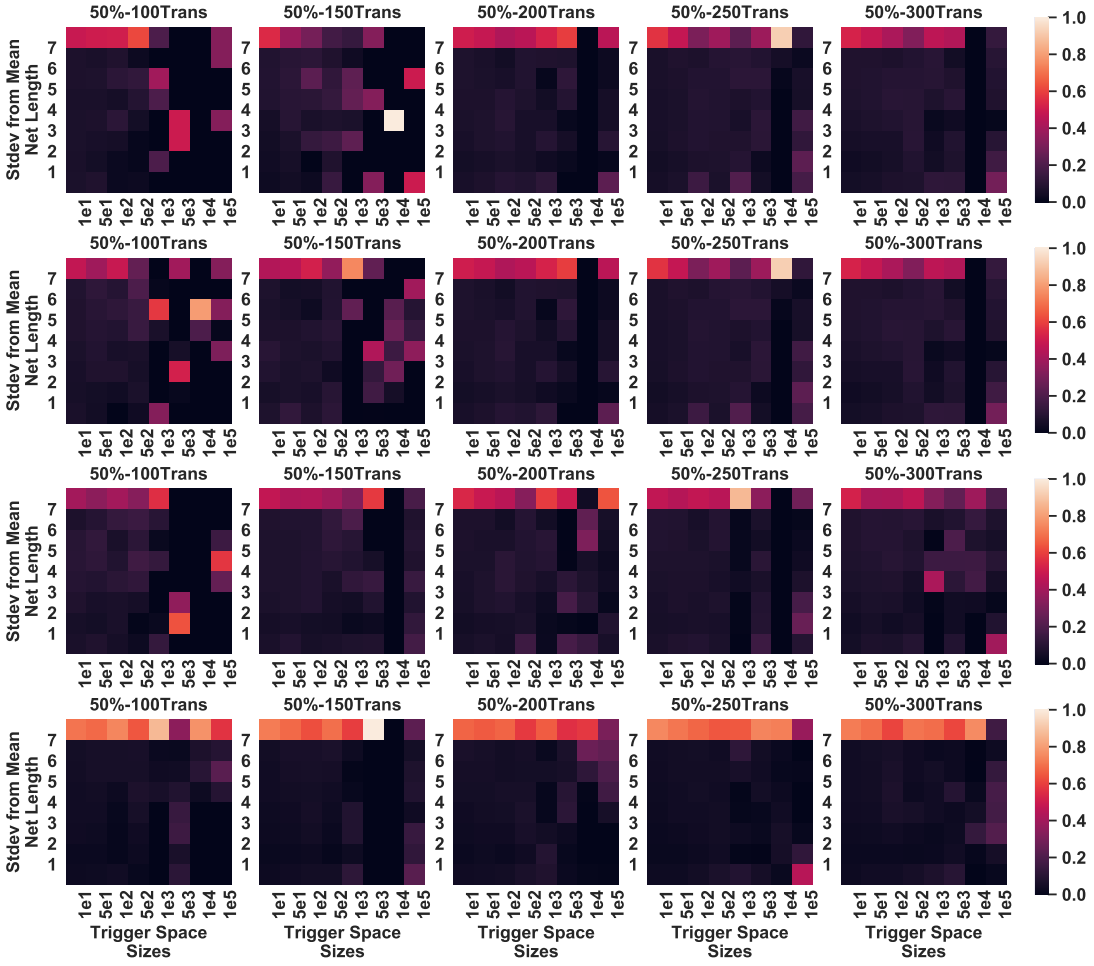

Figure A.1: **Route Distance Results for OR1200 at 50% Density.**

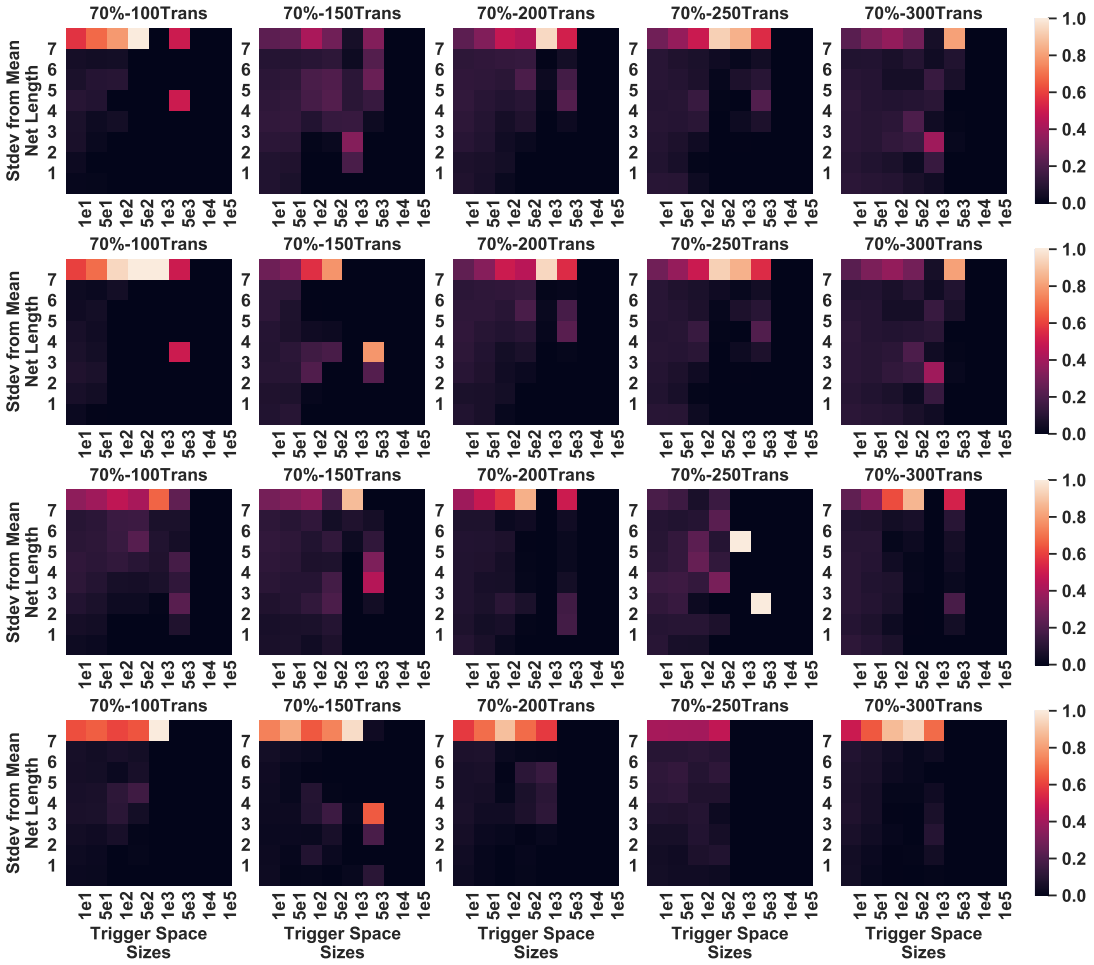

Figure A.2: **Route Distance Results for OR1200 at 70% Density.**

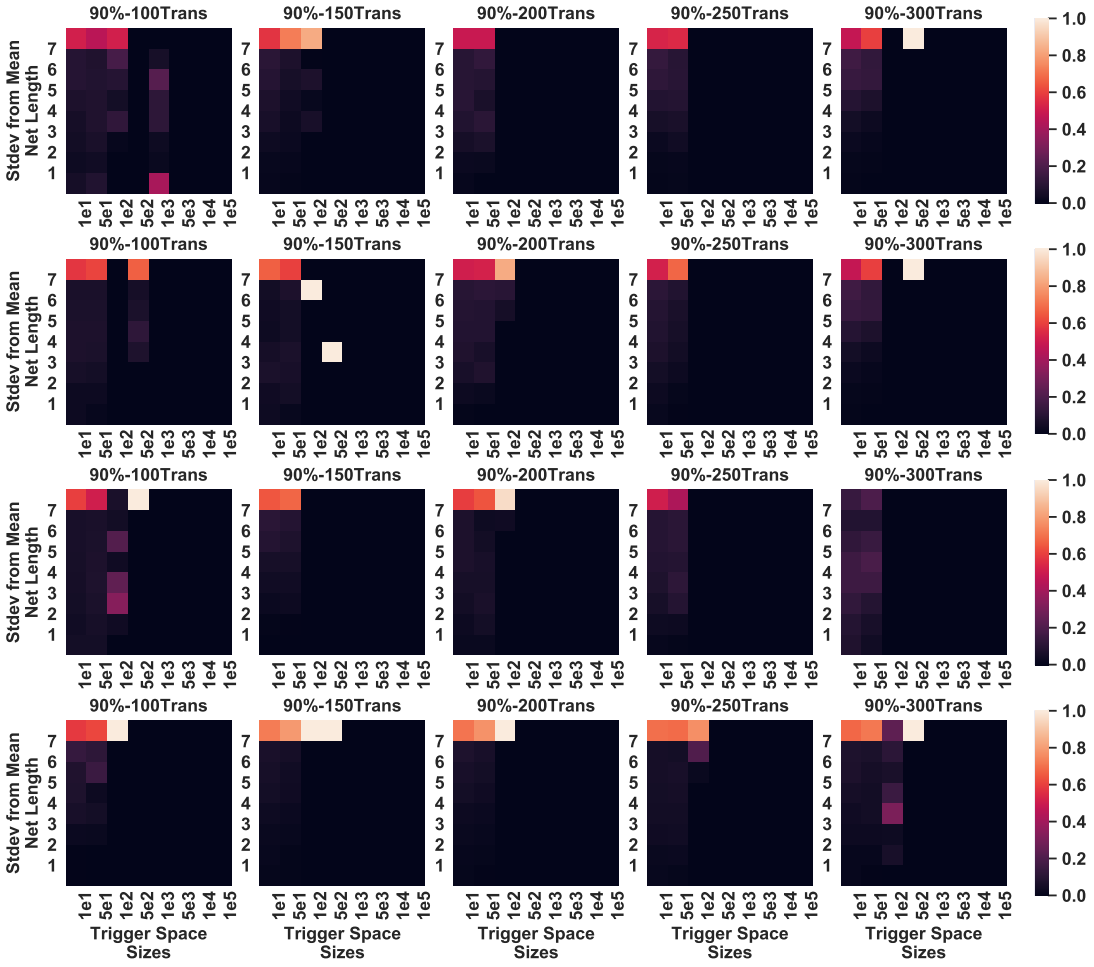

Figure A.3: **Route Distance Results for OR1200 at 90% Density.**

APPENDIX B

Descriptions of OpenTitan IP Blocks

AES

The OpenTitan AES core implements the Advanced Encryption Standard with key sizes of 128, 192, and 256 bits, and with the following cipher block modes: ECB, CBC, CFB, OFB, and CTR. Configuration settings, keys, and plaintext are delivered to the core through TileLink write operations to memory-mapped registers in a documented address range. Likewise, ciphertext is retrieved from the core through TileLink read operations. The core targets medium performance (one clock cycle per round of encryption). It implements a 128-bit wide data path—shared by encryption and decryption operations—that translates to encryption/decryption latencies of 12, 14, and 16 clock cycles per 128-bit plaintext block, in 128, 192, and 256 bit key modes, respectively. Of the cores I study, it is the second most complex in terms of LOC in both the hardware (HDL) and software domains (Table 6.2).

HMAC

The OpenTitan HMAC implements a SHA-256 hash message authentication code generator for the purpose of checking the integrity of incoming messages. The HMAC core

can operate in two modes: 1) SHA-256 mode only, or 2) HMAC mode. In the former mode, the core simply computes the SHA-256 hash of a provided message. In the latter mode, the core computes the HMAC (defined in RFC 2104 [89]) of a message using the SHA-256 hashing algorithm and a provided secret key. Regardless of mode, the SHA-256 engine operates on 512-bit message chunks at any given time, provided to the core through a message FIFO. Input messages can be read little- or big-endian and likewise, message digests can be stored in output registers either little- or big-endian. Configuration settings, input messages, HMAC keys, and operation commands are delivered to the core through TileLink write operations to memory-mapped registers. Likewise, message digests are retrieved from the core through TileLink read operations. In its current state, the core can hash a single 512-bit message in 80 clock cycles, and can compute its HMAC in 340 clock cycles. Of the cores I study, it is approximately half as complex as the AES core, in terms of LOC in both the hardware and software domains (Table 6.2).

KMAC

The OpenTitan KMAC core is similar to the HMAC core, except it implements a Keccak Message Authentication Code [83] and SHA-3 hashing algorithms. However, compared to the HMAC core, the KMAC core is more complex, as there are several more configurations. Specifically, there are many SHA-3 hashing functions that are supported—SHA3-224/256/384/512, SHAKE128/256, and cSHAKE128/256—and the $Keccak-f$ function (by default) operates on 1600 bits of internal state. Like the HMAC core, the KMAC core can simply compute hashes or message authentication codes depending on operation mode, and input messages/output digests can be configured to be read/stored in little- or big-endian. The time to process a single input message block is dominated by computing the $Keccak-f$ function, which takes 72 clock cycles for 1600 bits of internal state, in the current implementation of the core. Configuration settings, input messages, output digests, keys, and operation commands are all communicated to/from the core through TileLink

writes/reads to memory-mapped registers.

Of the cores I study, the KMAC core is the most complex, especially in the software domain (Table 6.2). The software model of the KMAC core contains almost 120k lines of C++ code. This is mostly an artifact of how Verilator maps dependencies between large registers and vectored signals: it creates large multidimensional arrays and maps each corresponding index at the word granularity. Fortunately, this artifact is optimized away during compilation, and the number of basic blocks in the DUT portion of the HSB is reduced.

RV-Timer

The OpenTitan RISC-V timer core is the simplest core I fuzz. It consists of a single 64-bit timer with 12-bit prescaler and an 8-bit step configurations. It can also generate system interrupts upon reaching a pre-configured time value. Like the other OpenTitan cores, the RV-Timer core is configured, activated, and deactivated via TileLink writes to memory-mapped registers.

APPENDIX C

Optimizing the Hardware Fuzzing Grammar

Recall, to facilitate widespread adoption of Hardware Fuzzing I design a generic testbench fuzzing harness that decodes a grammar and performs corresponding TL-UL bus transactions to exercise the DUT (Fig. 6.13). However, there are implementation questions surrounding how the grammar should be decoded (§6.4.2.4):

1. *How should I decode 8-bit opcodes when the opcode space defines less than* 2^8 *valid testbench actions?*

2. *How should I pack Hardware Fuzzing instruction frames that conform to my grammar?*

Opcode Formats

In its current state, I define three opcodes in my grammar that correspond to three actions my generic testbench can perform (Table 6.1): 1) **wait** one clock cycle, 2) TL-UL **read**, and 3) TL-UL **write**. However, I chose to represent these opcodes with a single byte (Fig. 6.4). Choosing a larger field than necessary has implications regarding the fuzzability of my grammar. In its current state, 253 of the 256 possible opcode values may be useless

depending on how they are decoded by the testbench. Therefore I propose, and empirically study, two design choices for decoding Hardware Fuzzing opcodes into testbench actions:

- **Constant**: *constant* values are used to represent each opcode corresponding to a single testbench action. Remaining opcode values are decoded as *invalid*, and ignored.
- **Mapped**: equal sized ranges of opcode values are *mapped* to valid testbench actions. No invalid opcode values exist.

Instruction Frame Formats

Of the three actions my testbench can perform—wait, read, and write—some require additional information. Namely, the TL-UL read action requires a 32-bit address field, and the TL-UL write action requires 32-bit data and address fields. Given this, there are two natural ways to decode Hardware Fuzzing instructions (Fig. 6.4):

- **Fixed**: a *fixed* instruction frame size is decoded regardless of the opcode. Address and data fields could go unused depending on the opcode.
- **Variable**: a *variable* instruction frame size is decoded. Address and data fields are only appended to opcodes that correspond to TL-UL read and write testbench actions. No address/data information goes unused.

Results

To determine the optimal Hardware Fuzzing grammar, I fuzz four OpenTitan IP blocks—the AES, HMAC, KMAC, and RV-Timer—for 24 hours using all combinations of opcode and instruction frame formats mentioned above. For each core I seed the fuzzer with 8–12 binary Hardware Fuzzing seed files (in the corresponding Hardware Fuzzing grammar) that correctly drive each core, with the exception of the RV-Timer core, which I seed with

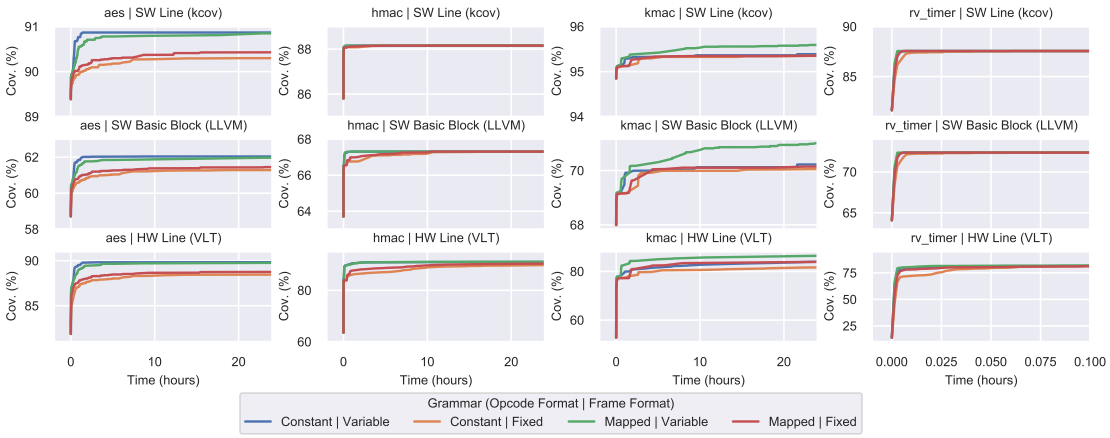

Figure C.1: **Coverage Convergence vs. Hardware Fuzzing Grammar.** Various software and hardware coverage metrics over fuzzing time across four OpenTitan [105] IP cores and hardware fuzzing grammar variations (§C). In the first row, we plot *line coverage* of the software models of each hardware core computed using `kcov`. In the second row, we plot *basic block coverage* computed using `LLVM`. In last row, we plot HDL line coverage (of the hardware itself) computed using `Verilator` [145]. From these results we formulate two conclusions: 1) coverage in the software domain correlates to coverage in the hardware domain, and 2) the Hardware Fuzzing grammar with *variable* instruction frames is best for greybox fuzzers that prioritize small test files.

a single wait operation instruction due to its simplicity. For each experiment, I extract and plot three DUT coverage metrics over fuzz times in Fig. C.1. These metrics include: 1) line coverage of the DUT software model, 2) basic block coverage of the same, and 3) line coverage of the DUT's HDL. Software line coverage is computed using `kcov` [79], software basic block coverage is computed using `LLVM` [96], and hardware line coverage is computed using `Verilator` [145]. Since I perform 10 repetitions of each fuzzing experiment, I average and consolidate each coverage time series into a single trace.

From these results I draw two conclusions. First, *variable* instruction frames seem to perform better than fixed frames, especially early in the fuzzing exploration. Since AFL prioritizes keeping test files small, I expect variable sized instruction frames to produce better results, since this translates to longer hardware test sequences, and therefore deeper possible explorations of the (sequential) state space. Second, the opcode type seems to make little difference, for most experiments, since there are only 256 possible values, a search space AFL can explore very quickly. Lastly, I point out that for simple cores, like the RV-Timer, Hardware Fuzzing is able to achieve ≈85% HDL line coverage in less than a minute (hence I do not plot the full 24-hour trace).

Key Insights:

1. Hardware Fuzzing instructions with **variable** frames are optimal for fuzzers that prioritize small input files, therefore resulting in longer temporal test *sequences*.
2. Increasing coverage in the software domain, translates to the hardware domain.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Accellera. Universal Verification Methodology (UVM). https://www.accellera.org/downloads/standards/uvm.

[2] Ronen Adato, Aydan Uyar, Mahmoud Zangeneh, Boyou Zhou, Ajay Joshi, Bennett Goldberg, and M Selim Unlu. Rapid mapping of digital integrated circuit logic gates via multi-spectral backside imaging. *arXiv:1605.09306*, 2016.

[3] Dakshi Agrawal, Selcuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan detection using IC fingerprinting. In *IEEE Symposium on Security and Privacy (SP)*, 2007.

[4] Paul Alcorn. Ice lake might arrive in june, according to leaked lenovo documents. https://www.tomshardware.com/news/lenovo-laptop-intel-ice-lake-10nm,38674.html.

[5] Yousra Alkabani and Farinaz Koushanfar. Designer's hardware trojan horse. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2008.

[6] Arm. Arm Cortex-M0. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0.

[7] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[8] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[9] Papa-Sidy Ba, Sophie Dupuis, Manikandan Palanichamy, Giorgio Di Natale, Bruno Rouzeyre, et al. Hardware trust through layout filling: a hardware trojan prevention technique. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016.

[10] Papa-Sidy Ba, Manikandan Palanichamy, Sophie Dupuis, Marie-Lise Flottes, Giorgio Di Natale, and Bruno Rouzeyre. Hardware trojan prevention using layout-level design approach. In *European Conference on Circuit Theory and Design (ECCTD)*, 2015.

[11] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference (DAC)*. IEEE, 2012.

[12] Halil B Bakoglu. Circuits, interconnections, and packaging for vlsi., 1990.

[13] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Electromagnetic circuit fingerprints for hardware Trojan detection. In *IEEE International Symposium on Electromagnetic Compatibility (EMC)*, 2015.

[14] Mainak Banga, Maheshwar Chandrasekar, Lei Fang, and Michael S Hsiao. Guided test generation for isolation and detection of embedded trojans in ics. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, 2008.

[15] Mainak Banga and Michael S Hsiao. A region based approach for the identification of hardware trojans. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2008.

[16] Mark Beaumont, Bradley Hopkins, and Tristan Newby. Hardware trojans-prevention, detection, countermeasures (a literature review). Technical report, Defence Science and Technology Organization Edinburgh (Australia), 2011.

[17] Georg T Becker, Francesco Regazzoni, Christof Paar, and Wayne P Burleson. Stealthy dopant-level hardware trojans. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2013.

[18] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In *Annual International Cryptology Conference*, 2008.

[19] John Blyler. Trends driving ip reuse through 2020, November 2017. `http://jbsystech.com/trends-driving-ip-reuse-2020/`.

[20] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[21] Duane Boning and Sani Nassif. Models of process variations in device and interconnect. *Design of high performance microprocessor circuits*, 2000.

[22] Mrinal Bose, Jongshin Shin, Elizabeth M Rudnick, Todd Dukes, and Magdy Abadir. A genetic approach to automatic bias generation for biased random instruction generation. In *Proceedings of the Congress on Evolutionary Computation*. IEEE, 2001.

[23] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[24] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[25] Cadence Design Systems. Innovus implementation system. https://www.cadence.com/content/cadence-www/global/en_US/home.html.

[26] Cadence Design Systems. JasperGold. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html.

[27] Cadence Design Systems. *Layer Map Files*. http://www-bsac.eecs.berkeley.edu/~cadence/tools/layermap.html.

[28] Cadence Design Systems. Xcelium Logic Simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.

[29] Cadence Design Systems. *LEF/DEF Language Reference*, 2009. http://www.ispd.cc/contests/14/web/doc/lefdefref.pdf.

[30] Yongming Cai, Zhiyong Wang, Rajen Dias, and Deepak Goyal. Electro optical terahertz pulse reflectometry—an innovative fault isolation tool. In *Electronic Components and Technology Conference (ECTC), 2010 Proceedings 60th*, 2010.

[31] Calma Company. *GDSII Stream Format Manual*, February 1987.

[32] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[33] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[34] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Hardware trojan: Threats and emerging solutions. In *IEEE High Level Design Validation and Test Workshop (HLDVT)*, 2009.

[35] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[36] Ming-Kun Chen, Cheng-Chi Tai, and Yu-Jung Huang. Nondestructive analysis of interconnection in two-die bga using tdr. *IEEE Transactions on Instrumentation and Measurement*, 2006.

[37] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[38] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[39] Marek Cieplucha. Metric-driven verification methodology with regression management. *Journal of Electronic Testing*, 2019.

[40] Ronald P Cocchi, James P Baukus, Lap Wai Chow, and Bryan J Wang. Circuit camouflage integration for hardware IP protection. In *ACM Design Automation Conference (DAC)*, 2014.

[41] lowRISC Contributors. OpenTitan: Comportability Definition and Specification, November 2020. `https://docs.opentitan.org/doc/rm/comportability_specification/`.

[42] MITRE Corporation. Cve details: Intel: Vulnerability statistics, August 2019. `https://www.cvedetails.com/vendor/238/Intel.html`.

[43] Jason Cross. Inside apple's A13 bionic system-on-chip, October 2019. `https://www.macworld.com/article/3442716/inside-apples-a13-bionic-system-on-chip.html`.

[44] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[45] William C Elmore. The transient response of damped linear networks with particular regard to wideband amplifiers. *Journal of Applied Physics*, 1948.

[46] Keoni Everington. TSMC starts work on US$19.6 billion 3nm fab in s. taiwan, October 2019. `https://www.taiwannews.com.tw/en/news/3805032`.

[47] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *the 40th annual Design Automation Conference (DAC)*, 2003.

[48] Domenic Forte, Chongxi Bao, and Ankur Srivastava. Temperature tracking: An innovative run-time approach for hardware trojan detection. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013.

[49] Kelson Gent and Michael S Hsiao. Fast multi-level test generation at the rtl. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016.

[50] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 2012.

[51] Lawrence H Goldstein and Evelyn L Thigpen. SCOAP: Sandia controllability/observability analysis program. In *ACM Design Automation Conference (DAC)*, 1980.

[52] Google LLC. RISCV-DV. https://github.com/google/riscv-dv.

[53] Preeti Gupta. 7nm power issues and solutions, November 2016. https://semiengineering.com/7nm-power-issues-and-solutions/.

[54] Onur Guzey and Li-C Wang. Coverage-directed test generation through automatic constraint extraction. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2007.

[55] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *the 37th annual International Symposium on Computer Architecture (ISCA)*, 2010.

[56] Leonard A Hayden and Vijai K Tripathi. Characterization and modeling of multiple line interconnections from time domain measurements. *IEEE Transactions on Microwave Theory and Techniques*, 1994.

[57] Jesse Hertz and Tim Newsham. ProjectTriforce: AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL.

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

[59] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. SPECS: A lightweight runtime mechanism for protecting software from security-critical processor bugs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[60] Simon Hollis and Simon W Moore. Rasp: an area-efficient, on-chip network. In *2006 International Conference on Computer Design*, pages 63–69. IEEE, 2006.

[61] Simon J Hollis. Pulse generation for on-chip data transmission. In *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 303–310. IEEE, 2009.

[62] Yumin Hou, Hu He, Kaveh Shamsi, Yier Jin, Dong Wu, and Huaqiang Wu. R2D2: Runtime reassurance and detection of A2 trojan. In *International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2018.

[63] MS Hrishikesh, Norman P Jouppi, Keith I Farkas, Doug Burger, Stephen W Keckler, and Premkishore Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *IEEE International Symposium on Computer Architecture (ISCA)*, 2002.

[64] Ching-Wen Hsue and Te-Wen Pan. Reconstruction of nonuniform transmission lines from time-domain reflectometry. *IEEE Transactions on Microwave Theory and Techniques*, 1997.

[65] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. PANGOLIN: Incremental hybrid fuzzing with polyhedral path abstraction. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[66] John F Hughes and James D Foley. *Computer graphics: principles and practice*. Pearson Education, 2014.

[67] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DIFUZZRTL: Differential fuzz testing to find cpu bugs. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[68] Frank Imeson, Ariq Emtenan, Siddharth Garg, and Mahesh Tripunitara. Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation. In *USENIX Security Symposium*, 2013.

[69] Frank Imeson, Saeed Nejati, Siddharth Garg, and Mahesh Tripunitara. Non-deterministic timers for hardware trojan activation (or how a little randomness can go the wrong way). In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, 2016.

[70] SiFive Inc. SiFive: TileLink Specification, November 2020. Version 1.8.0.

[71] Intel Corporation. Microprocessor quick reference guide. `https://www.intel.com/pressroom/kits/quickreffam.htm#i486`.

[72] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. Introducing xcs to coverage directed test generation. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2011.

[73] Peter Jamieson, Kenneth B Kent, Farnaz Gharibian, and Lesley Shannon. Odin II-an open-source verilog HDL synthesis tool for CAD research. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2010.

[74] Yier Jin, Nathan Kupp, and Yiorgos Makris. DFTT: Design for trojan test. In *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2010.

[75] Yier Jin and Yiorgos Makris. Hardware trojan detection using path delay fingerprint. In *IEEE Workshop on Hardware-Oriented Security and Trust (HOST)*, 2008.

[76] James Johnson. gramfuzz. `https://github.com/d0c-s4vage/gramfuzz`.

[77] Jing-Yang Jou and Chien-Nan Jimmy Liu. Coverage analysis techniques for hdl design validation. *Proceedings of Asia Pacific CHip Design Languages*, 1999.

[78] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 2018.

[79] Simon Kagstrom. kcov. `https://github.com/SimonKagstrom/kcov`.

[80] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. Replacing testing with formal verification in intel core i7 processor execution engine validation. In *International Conference on Computer Aided Verification (CAV)*. Springer, 2009.

[81] Ramesh Karri, Jeyavijayan Rajendran, Kurt Rosenfeld, and Mohammad Tehranipoor. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer*, 2010.

[82] Shane Kelly, Xuehui Zhang, Mohammed Tehranipoor, and Andrew Ferraiuolo. Detecting hardware trojans using on-chip sensors in an asic design. *Journal of Electronic Testing*, 31(1):11–26, 2015.

[83] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. Technical report, National Institute of Standards and Technology, 2016.

[84] Tae Kim. Intel's alleged security flaw could cost chipmaker a lot of money, Bernstein says. `https://www.cnbc.com/2018/01/03/intels-alleged-security-flaw-could-cost-chipmaker-a-lot-of-money-bernstein.html`.

[85] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and implementing malicious hardware. In *Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[86] Angus I Kingon, Jon-Paul Maria, and SK Streiffer. Alternative dielectrics to silicon dioxide for memory and logic devices. *Nature*, 2000.

[87] Christopher H Kingsley and Balmukund K Sharma. Method and apparatus for identifying flip-flops in hdl descriptions of circuits without specific templates, 1998. US Patent 5,854,926.

[88] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[89] H. Krawczyk, M. Bellare, and M. Bellare. HMAC: Keyed-hashing for message authentication. RFC 2104, RFC Editor, February 1997.

[90] Raghavan Kumar, Philipp Jovanovic, Wayne Burleson, and Ilia Polian. Parametric trojans for fault-injection attacks on cryptographic hardware. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2014.

[91] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: coverage-directed fuzz testing of RTL on FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018.

[92] Mark Lapedus. 10nm versus 7nm, April 2016. `https://semiengineering.com/10nm-versus-7nm/`.

[93] Mark Lapedus. Battling fab cycle times, February 2017. `https://semiengineering.com/battling-fab-cycle-times/`.

[94] Mark Lapedus. Big trouble at 3nm, June 2018. `https://semiengineering.com/big-trouble-at-3nm/`.

[95] Mark Lapedus. GF puts 7nm on hold, August 2018. `https://semiengineering.com/gf-puts-7nm-on-hold/`.

[96] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, USA, 2004.

[97] Jie Li and John Lach. At-speed delay characterization for IC authentication and trojan horse detection. In *IEEE Workshop on Hardware-Oriented Security and Trust (HOST)*, 2008.

[98] Patrick S Li, Adam M Izraelevitz, and Jonathan Bachrach. Specification for the FIRRTL language. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9*, 2016.

[99] Wenchao Li, A. Gascon, P. Subramanyan, Wei Yang Tan, A. Tiwari, S. Malik, N. Shankar, and S.A. Seshia. Wordrev: Finding word-level structures in a sea of bit-level gates. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2013.

[100] Jun Jun Lim, Nor Adila Johari, Subhash C Rustagi, and Narain D Arora. Characterization of interconnect process variation in cmos using electrical measurements and field solver. *IEEE Transactions on Electron Devices*, 2014.

[101] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2009.

[102] Timothy Linscott, Pete Ehrett, Valeria Bertacco, and Todd Austin. SWAN: mitigating hardware trojans with design ambiguity. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[103] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security)*, 2018.

[104] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. `https://llvm.org/docs/LibFuzzer.html`.

[105] lowRISC. Opentitan. `https://opentitan.org/`.

[106] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. ASIC clouds: Specializing the datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 178–190. IEEE, 2016.

[107] Mentor Graphics. ModelSim. `https://www.mentor.com/products/fv/modelsim/`.

[108] MIT Lincoln Laboratory. Common evaluation platform. `https://github.com/mit-ll/CEP`.

[109] MIT Lincoln Laboratory. Common evaluation platform. `https://github.com/mit-ll/CEP/tree/d19a5de3dc32d58b535f52fc9aa2cd70f95107e1`.

[110] MIT Lincoln Laboratory. GDS2-Score. `https://github.com/mit-ll/gds2-score`.

[111] MIT Lincoln Laboratory. Nemo. `https://github.com/mit-ll/nemo`.

[112] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[113] Gordon E Moore. Cramming more components onto integrated circuits, 1965.

[114] Mozilla Security. Dharma: A generation-based, context-free grammar fuzzer. `https://www.overleaf.com/project/5e163844e63c070001079faa`.

[115] Michael Nagel, Alexander Michalski, and Heinrich Kurz. Contact-free fault location and imaging with on-chip terahertz time-domain reflectometry. *Optics Express*, 2011.

[116] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[117] Seetharam Narasimhan, Xinmu Wang, Dongdong Du, Rajat Subhra Chakraborty, and Swarup Bhunia. TeSR: A robust temporal self-referencing approach for hardware trojan detection. In *IEEE Symposium on Hardware-Oriented Security and Trust (HOST)*, 2011.

[118] Tony Nowatzki, Vinay Gangadhan, Karthikeyan Sankaralingam, and Greg Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[119] C Odegard and C Lambert. Comparative tdr analysis as a packaging fa tool. In *ISTFA 1999: 25 th International Symposium for Testing and Failure Analysis*, 1999.

[120] University of California. Risc-v gnu compiler toolchain. `https://github.com/riscv/riscv-gnu-toolchain`.

[121] OpenCores.org. OpenRISC OR1200 processor. `https://github.com/openrisc/or1200`.

[122] OpenCores.org. Openrisc or1k tests. `https://github.com/openrisc/or1k-tests/tree/master/native/or1200`.

[123] OpenCores.org. Or1k-elf toolchain. `https://openrisc.io/newlib/`.

[124] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *USENIX Security Symposium*, 2020.

[125] V.A. Patankar, A. Jain, and R.E. Bryant. Formal verification of an ARM processor. In *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pages 282–287, 1999.

[126] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with flex. *University of California*, 2007.

[127] Peach Tech. Peach Fuzzing Platform. `https://www.peach.tech/products/peach-fuzzer/`.

[128] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[129] Dan L Philen, Ian A White, Jane F Kuhl, and Stephen C Mettler. Single-mode fiber otdr: Experiment and theory. *IEEE Transactions on Microwave Theory and Techniques*, 1982.

[130] Andrew Piziali. *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.

[131] Miodrag Potkonjak, Ani Nahapetian, Michael Nelson, and Tammara Massey. Hardware trojan horse detection using gate-level characterization. In *ACM/IEEE Design Automation Conference (DAC)*, 2009.

[132] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Network and Distributed Systems Security Symposium (NDSS)*, 2017.

[133] UC Berkeley Architecture Research. Pipelined FFT. https://github.com/ucb-art/fft.

[134] UC Berkeley Architecture Research. RISC-V Sodor. https://github.com/ucb-bar/riscv-sodor.

[135] Masoud Rostami, Farinaz Koushanfar, Jeyavijayan Rajendran, and Ramesh Karri. Hardware security: Threat models and metrics. In *IEEE International Conference on Computer-Aided Design (ICCD)*, 2013.

[136] H. Salmani, M. Tehranipoor, and R. Karri. On design vulnerability analysis and trust benchmarks development. In *IEEE International Conference on Computer Design (ICCD)*, 2013.

[137] Hassan Salmani. COTD: Reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist. *IEEE Transactions on Information Forensics and Security*, 2017.

[138] Hassan Salmani and Mohammed Tehranipoor. Analyzing circuit vulnerability to hardware trojan insertion at the behavioral level. In *IEEE Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2013.

[139] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity Development (SecDev)*. IEEE, 2016.

[140] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. In *USENIX Security Symposium*, 2017.

[141] Sophia Shao and Emma Wang. Die photo analysis. http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/.

[142] Yuriy Shiyanovskii, F Wolff, Aravind Rajendran, C Papachristou, D Weyer, and W Clay. Process reliability based trojans through NBTI and HCI effects. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2010.

[143] SiFive. SiFive Blocks. https://github.com/sifive/sifive-blocks.

[144] D Smolyansky. Electronic package fault isolation using tdr. *ASM International*, 2004.

[145] Wilson Snyder. verilator. https://www.veripool.org/wiki/verilator.

[146] PI Somlo and DL Hollway. Microwave locating reflectometer. *Electronics Letters*, 1969.

[147] Ed Sperling. Design rule complexity rising, April 2018. https://semiengineering.com/design-rule-complexity-rising/.

[148] Giovanni Squillero. Microgp—an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 2005.

[149] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed Systems Security Symposium (NDSS)*, 2016.

[150] Cynthia Sturton, Matthew Hicks, David Wagner, and Samuel T King. Defeating uci: Building stealthy and malicious hardware. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2011.

[151] Pramod Subramanyan, Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea, and Sharad Malik. Reverse engineering digital circuits using functional analysis. In *Proceedings of the ACM Conference on Design, Automation and Test in Europe (DATE)*, 2013.

[152] Takeshi Sugawara, Daisuke Suzuki, Ryoichi Fujii, Shigeaki Tawa, Ryohei Hori, Mitsuru Shiozaki, and Takeshi Fujino. Reversing stealthy dopant-level circuits. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2014.

[153] James Sutherland. As edge speeds increase, wires become transmission lines. *EDN*, 1999.

[154] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[155] Robert Swiecki. honggfuzz. `https://honggfuzz.dev/`.

[156] Synopsys. VC Formal. `https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html`.

[157] Synopsys. VCS. `https://www.synopsys.com/verification/simulation/vcs.html`.

[158] Dean Takahashi. Intel will invest in factories and manufacture chips for other companies. `https://venturebeat.com/2021/03/23/intel-will-invest-in-factories-and-manufacture-chips-for-other-companies/`.

[159] Serdar Tasiran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 2001.

[160] MY Tay, L Cao, M Venkata, L Tran, W Donna, W Qiu, J Alton, PF Taday, and M Lin. Advanced fault isolation technique using electro-optical terahertz pulse reflectometry. In *Physical and Failure Analysis of Integrated Circuits (IPFA), 2012 19th IEEE International Symposium on the*, 2012.

[161] Jake Taylor. Kaze: an HDL embedded in Rust, November 2020. `https://docs.rs/kaze/0.1.13/kaze/`.

[162] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers*, 2010.

[163] Marat Teplitsky, Amit Metodi, and Raz Azaria. Coverage driven distribution of constrained random stimuli. In *Proceedings of the Design and Verification Conference (DVCon)*, 2015.

[164] TeraView. *Electro Optical Terahertz Pulse Reflectometry: The world's fastest and most accurate fault isolation system.* `https://teraview.com/eotpr/`.

[165] Texplained. ChipJuice. `https://www.texplained.com/about-us/chipjuice-software`.

[166] The GNU Project. Bison. `https://www.gnu.org/software/bison/`.

[167] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 109–120, 2009.

[168] Timothy Trippel. Bomberman, December 2020. `https://github.com/timothytrippel/bomberman`.

[169] Timothy Trippel, Kang G. Shin, Kevin B. Bush, and Matthew Hicks. ICAS: an extensible framework for estimating the susceptibility of ic layouts to additive trojans. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[170] Timothy Trippel, Kang G. Shin, Kevin B. Bush, and Matthew Hicks. T-TER: Defeating A2 Trojans with Targeted Tamper-Evident Routing. *ArXiv*, 2020.

[171] Timothy Trippel, Kang G. Shin, Kevin B. Bush, and Matthew Hicks. Bomberman: Defining and Defeating Hardware Ticking Timebombs at Design-time. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[172] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing Hardware Like Software. *ArXiv*, 2021.

[173] TSMC. Tsmc fabrication schedule — 2019, April 2019. `https://www.mosis.com/db/pubf/fsched?ORG=TSMC`.

[174] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security)*, 2018.

[175] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security'18)*, 2018.

[176] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[177] Potential Ventures. cocotb. `https://github.com/cocotb/cocotb`.

[178] Denys Vlasenko. Busybox. `https://www.busybox.net/`.

[179] Martin Vuagnoux. Autodaf´e: an act of software torture. `http://autodafe.sourceforge.net/tutorial/index.html`.

[180] Dmitry Vyukov. syzkaller. `https://github.com/google/syzkaller`.

[181] Adam Waksman, Jeyavijayan Rajendran, Matthew Suozzo, and Simha Sethumadhavan. A red team/blue team assessment of functional analysis methods for malicious circuit identification. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014.

[182] Adam Waksman and Simha Sethumadhavan. Tamper evident microprocessors. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[183] Adam Waksman and Simha Sethumadhavan. Silencing hardware backdoors. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[184] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[185] Fanchao Wang, Hanbin Zhu, Pranjay Popli, Yao Xiao, Paul Bodgan, and Shahin Nazarian. Accelerating coverage directed test generation for functional verification: A neural network-based framework. In *Proceedings of the Great Lakes Symposium on VLSI*, 2018.

[186] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[187] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.

[188] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[189] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[190] Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. Detecting malicious inclusions in secure hardware: Challenges and solutions. In *IEEE Workshop on Hardware-Oriented Security and Trust (HOST)*, 2008.

[191] Xinmu Wang, Seetharam Narasimhan, Aswin Krishna, Tatini Mal-Sarkar, and Swarup Bhunia. Sequential hardware trojan: Side-channel aware design and placement. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*, 2011.

[192] Stephen Williams. Icarus Verilog. `http://iverilog.icarus.com/`.

[193] Clifford Wolf. Picorv32. `https://github.com/cliffordwolf/picorv32#cycles-per-instruction-performance`.

[194] Francis Wolff, Chris Papachristou, Swarup Bhunia, and Rajat S Chakraborty. Towards trojan-free trusted ICs: Problem analysis and detection scheme. In *ACM Conference on Design, Automation and Test in Europe (DATE)*, 2008.

[195] Kan Xiao and Mohammed Tehranipoor. BISA: Built-in self-authentication for preventing hardware trojan insertion. In *IEEE Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013.

[196] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog malicious hardware. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

[197] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[198] Jun Yuan, Carl Pixley, Adnan Aziz, and Ken Albin. A framework for constrained functional verification. In *International Conference on Computer Aided Design (ICCAD)*. IEEE, 2003.

[199] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In {*USENIX*} *Security Symposium*, 2018.

[200] Michael Zalewski. afl-fuzz: making up grammar with a dictionary in hand. `https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html`.

[201] Michael Zalewski. American fuzzy lop. `https://lcamtuf.coredump.cx/afl/`.

[202] Jie Zhang, Feng Yuan, Linxiao Wei, Yannan Liu, and Qiang Xu. VeriTrust: Verification for hardware trust. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2015.

[203] Jie Zhang, Feng Yuan, and Qiang Xu. DeTrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[204] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018.

[205] Rui Zhang, Natalie Stanley, Christopher Griggs, Andrew Chi, and Cynthia Sturton. Identifying security critical properties for the dynamic verification of a processor. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[206] Rui Zhang and Cynthia Sturton. A recursive strategy for symbolic execution to find exploits in hardware designs. In *ACM SIGPLAN International Workshop on Formal Methods and Security (FSM)*, 2018.

[207] Rui Zhang and Cynthia Sturton. Transys: Leveraging common security properties across hardware designs. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[208] Xuehui Zhang and Mohammad Tehranipoor. Ron: An on-chip ring oscillator network for hardware trojan detection. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[209] Boyou Zhou, Ronen Adato, Mahmoud Zangeneh, Tianyu Yang, Aydan Uyar, Bennett Goldberg, Selim Unlu, and Ajay Joshi. Detecting hardware trojans using backside optical imaging of embedded watermarks. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[210] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *USENIX Security Symposium*, 2020.