Producing Trustworthy Hardware Using Untrusted Components, Personnel and Resources

Adam Waksman

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2014

ABSTRACT

Producing Trustworthy Hardware Using Untrusted Components, Personnel and Resources

Adam Waksman

Computer security is a full-system property, and attackers will always go after the weakest link in a system. In modern computing systems, the hardware supply chain is an obvious and vulnerable point of attack. The ever-increasing complexity of hardware systems, along with the globalization of the hardware supply chain, has made it unreasonable to trust hardware. Hardware-based attacks, known as backdoors, are easy to implement and can undermine the security of systems built on top of compromised hardware. Operating systems and other software can only be secure if they can trust the underlying hardware systems.

The full supply chain for creating hardware includes multiple processes, which are often addressed in disparate threads of research, but which we consider as one unified process. On the front-end side, there is the soft design of hardware, along with validation and synthesis, to ultimately create a netlist, the document that defines the physical layout of hardware. On the back-end side, there is a physical fabrication process, where a chip is produced at a foundry from a supplied netlist, followed in some cases by post-fabrication testing. Producing a trustworthy chip means securing this process from the early design stages through to the post-fabrication tests.

We propose, implement and analyze a series of methods for making the hardware supply chain resilient against a wide array of known and possible attacks. These methods allow for the design and fabrication of hardware using untrustworthy personnel, designs, tools and resources, while protecting the final product from large classes of attacks, some known previously and some discovered and taxonomized in this work. The overarching idea in this work is to take a full-process view of the hardware supply chain. We begin by securing the hardware design and synthesis processes using a defense-in-depth approach. We combine

this work with foundry-side techniques to prevent malicious modifications and counterfeiting. Finally, we apply novel attestation techniques to ensure that hardware is trustworthy when it reaches users.

For our design-side security approach, we use defense-in-depth because in practice any security method can potentially be subverted, and defense-in-depth is the best way to handle that assumption. Our approach involves three independent steps. The first is a functional analysis tool (called **FANCI**), applied statically to designs during the coding and validation stages, to remove any malicious circuits. The second step is to include physical security circuits that operate at runtime. These circuits, which we call trigger obfuscation circuits, scramble data at the microarchitectural level so that any hardware backdoors remaining in the design cannot be triggered at runtime. The third and final step is to include a runtime monitoring system that detects any backdoor payloads that might have been achieved despite the previous two steps. We design two different versions of this monitoring system. The first, **TrustNet**, is extremely lightweight and protects against an important class of attacks called emitter backdoors. The second, **DataWatch**, is slightly more heavyweight (though still efficient and low overhead), can catch a wider variety of attacks and can be adapted to protect against nearly any type of digital payload. We taxonomize the types of attacks that are possible against each of the three steps of our defense-in-depth system and show that each defense provides strong coverage with low (or negligible) overheads to performance, area and power consumption.

For our foundry-side security approach we develop the first foundry-side defense system that is aware of design-side security. We create a power-based side-channel, called a *beacon*. This beacon is essentially a benign backdoor. It can be turned on by a special key (not provided to the foundry), allowing for security attestation during post-fabrication testing. By embedding this beacon into the hardware design itself, the beacon requires neither keys nor storage and as such exists in the final chip purely by virtue of existing in the netlist. We further obfuscate the netlist itself, rendering the task of reverse engineering the beacon (for a foundry-side adversary) intractable. Both the inclusion of the beacon and the obfuscation process add little to area and power costs and have no impact on performance.

All together, these methods provide a foundation on which hardware security can be

developed and enhanced. They are low overhead and practical, making them suitable for inclusion in next generation hardware. Moving forward, the criticality of having trustworthy hardware can only increase. Ensuring that the hardware supply chain can be trusted in the face of sophisticated adversaries is vital. Both hardware design and hardware fabrication are increasingly international processes, and we believe that continuing with this unified approach is the correct path for future research. In order for companies and governments to place trust in mission-critical hardware, it is necessary for hardware to be certified as secure and trustworthy. The methods we propose can be the first steps toward making this certification a reality.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Over the course of five years at Columbia University, there are plenty of people to thank for their support, advice and insights. I should first thank my advisor, Dr. Simha Sethumadhavan, who has been involved in each step of this process. The environment he set up was one in which I felt confident focusing on relevant research from day one, and his desire to always be pushing on new threads of research has been beneficial and encouraging. He also has placed a tremendous focus on clarity of communication and writing so as to produce research that is not only important but is also understood by someone. Hopefully this document is not an exception.

I would next like to thank the colleagues I have interacted with at Columbia, which includes all of my coauthors and all of the Professors who have helped in supporting me. Thank you to Dr. Salvatore Stolfo, Dr. Steven Bellovin and Dr. Allison Lewko for serving on my defense committee, as well as to Dr. Sam King of the University of Illinois at Urbana-Champaign. Thank you as well to Dr. Angelos Keromytis, Dr. Martha Kim and all the other Professors who have read my work, attended my talks or discussed research with me. Thank you to Dr. Janet Kayfetz of the University of California Santa Barbara for helping me with scientific writing and presentation skills. And thank you to all other members of CASTL, CSL and CRF at Columbia who have provided academic and technical support to my research endeavors.

Finally, thank you to my parents and the rest of my family, who I expect fought their way through the first 16 pages of this document to reach this page. It is obviously a privileged position to be in to be able to spend nine years getting an education, and my family has been entirely supportive throughout.

Chapter 1

Introduction to Hardware Security

The perception of computer hardware has changed radically over the past few years. For much of the history of computing, hardware was used as the root of trust, a static and immutable source of secure functionality, on top of which the entire computing stack was built. This perception has changed due to the realization that hardware is just as vulnerable to attacks and security violations as the software that is built on top of it. While the vulnerability of hardware was perhaps always present, both the extent of the vulnerability and the awareness of it have grown palpably due to the nature of modern hardware design.

Hardware development is now an incredibly global process. The creation of any individual system will often involve the use of components and intellectual property from a wide variety of corporations around the world. On top of that, fabrication processes are often performed in different countries and by different corporations than the ones creating and synthesizing designs. The massive scale and complexity of modern hardware has all but ended the era when a small group of trusted personnel could design a chip. Instead, even for hardware that is vital to corporate or national security, the same commercial processes are used, meaning that security rests on the assumed trust in a variety of corporations around the world.

This situation has enormous ramifications for both large corporations and national governments. In both cases, these large organizations need to trust the computing systems on which they rely every day. However, in order to keep up with modern technology, they are heavily incentivized to participate in the global computing economy. Only by developing

a rigorous notion of security for the global hardware supply chain can these organizations hope to regain trust in the devices that they use.

Thesis & Contributions: We claim that it is possible to build practical, trustworthy hardware systems while using untrusted resources, tools and personnel. It is not necessary to assume trust in personnel, such as designers and microarchitects, nor is it necessary to assume trust in individual components that are combined to build a complete system. By taxonomizing the space of possible attacks and malicious agents, we construct a model of the hardware supply chain and identify the locations where compromises can occur. We then construct an end-to-end security methodology that allows for the combination of untrusted components and reliance on untrusted personnel to create a full hardware system that can be trusted. Since it will never be possible to fully trust all personnel and resources being used in this increasingly global supply chain, this holistic approach, using the philosophy of building trustworthy systems out of untrusted pieces, is the only approach that can lead to a satisfying solution.

1.1 How is Hardware Built?

The supply chain for creating modern hardware systems relies on several processes, all of which must be trustworthy and reliable. In the broadest sense, the process splits into two halves: design and fabrication. Design is the act of creating the definition of what the hardware is supposed to do. This involves all of the 'soft' components of the process and can also be thought of as the 'front end' of the process. Fabrication is the physical half of the process, where the design is used to create a physical product. This involves the 'hard' aspects of the process and can also be thought of as the 'back end' of the process.

More specifically, the design half includes a variety of processes that turn a high-level specification into a low-level design. The specification process defines the semantics and functionality of the desired hardware, including an agreement to an Instruction Set Architecture (ISA) and the subsequent designing of the architecture of the hardware. Then the core design phase is the designing of the lower level microarchitecture and the transformation of the design into precise code, usually in a Hardware Description Language (HDL).

These languages are often but not always Register Transfer Level (RTL) languages. After design comes a validation and verification process. Validation involves the testing of properties of the hardware using test inputs. Verification involves verifying or proving correctness properties of the hardware or of certain subcomponents. Verification can range in scope and quality, with some designers using only validation tests and others using fully formal verification, where the correctness of the entire design is proven. Formal verification is an extreme option that is often not possible. Design and validation are often done hand-in-hand, with several iterations going back and forth, with each iteration uncovering bugs and improving the design.

After the front-end processes have been completed, the validated and/or verified design undergoes physical synthesis and layout, where the functional design is translated first into a functional gatelist and then finally into a netlist layout, which is a blueprint for the physical layout of the chip. These translations are usually done by automated tools but in some cases can be performed by custom circuit engineers. This netlist is then used at a fabrication facility (or foundry) to produce a physical chip, a process often refered to as tape-out. After tape-out, a chip may undergo some degree of final testing, usually to check for manufacturing defects. In any given manufacturing run, a substantial portion of the chips will not work correctly, even with the best of intentions (*i.e.* the yield is never 100%).

1.2 Motivating End-to-End Security

The core philosophy of this work is to develop security in an end-to-end fashion, taking into account all agents and processes. This means an outlook on security that does not simply create trust in a specific piece or component but instead necessitates a global view of hardware development and security protocols that can coherently handle attacks in all phases. Given the incredible economic impact of hardware on the modern global economy, we must assume that potential attackers have tremendous resources at their disposal. This means that it must be considered possible that attackers can form conspiracies across multiple steps of the process and that they can use extremely sophisticated attacks.

As a result of this situation, we must develop methods that can secure hardware from

specification through to fabrication. These methods must be aware of each other and must not conflict with each other. We also must assume that attackers know all details of our defenses and can invest substantial resources toward violating the axioms and assumptions of these methods.

1.3 Novel Methods for Hardware Security

In this dissertation, we propose, implement, analyze and evaluate a set of methods for making the hardware supply chain resilient against a wide array of known and possible attacks. We begin by securing the hardware design and synthesis processes using a *defense-in-depth approach*. We combine this half of our work with novel foundry-side techniques to prevent both malicious modifications and counterfeiting. In the final step, we use a new method for attestation to ensure that hardware is trustworthy when it reaches consumers.

The reason we use a defense-in-depth approach is that in practice any security method can be subverted given unlimited resources. Defense-in-depth is the best way to handle that problem. Using a single defense method that is assumed to be impenetrable is often a recipe for failure. Our approach involves three independent steps, and an attacker must violate the axioms of all three simultaneously to achieve a malicious goal.

The first method is a functional analysis tool (called **FANCI**, which stands for Functional Analysis for Nearly-Unused Circuit Identification), applied statically to designs during the coding and validation stages to directly remove any malicious circuits. The second step is to include physical security circuits that operate at runtime. These circuits, which we call trigger obfuscation circuits, scramble data at the microarchitectural level so that any malicious hardware circuits (commonly referred to as backdoors), remaining in the design cannot be triggered at runtime. The third and final step is to include a runtime monitoring system that detects any backdoor payloads that might have been achieved despite the previous two methods. We design two different versions of this monitoring system. The first, **TrustNet**, is extremely lightweight, while the second, **DataWatch**, is slightly more heavyweight and covers a wider set of attacks. We further taxonomize the types of attacks that are possible against each of the three steps of our defense-in-depth system and show that

each defense provides strong coverage with low (or negligible) overheads to performance, area and power consumption.

For our foundry-side security approach (the other half of our solution), we develop the first foundry-side defense system that is explicitly aware of design-side security. We create a power-based side-channel, called a beacon. This beacon can be turned on by a special key (not provided to the foundry and not stored on chip), allowing for security attestation during post-fabrication testing. By placing this beacon in the design itself, the beacon requires neither keys nor storage and as such exists in the final chip purely by virtue of existing in the netlist. We further obfuscate the netlist, rendering the task of reverse engineering the beacon intractable. The inclusion of the beacon and the obfuscation process add little to area and power costs and have no direct impact on performance.

All together, our methods provide a foundation on which hardware security can be developed and enhanced. They are low overhead, practical and effective, making them suitable for inclusion in next generation hardware. Moving forward, the criticality of having trustworthy hardware can only increase. Ensuring that the hardware supply chain can be trusted in the face of sophisticated adversaries is vital. Both hardware design and hardware fabrication are increasingly international processes, and we believe that continuing with a unified approach is the correct path for future research. In order for companies, militaries and governments to place trust in mission-critical hardware, we believe it is necessary for hardware to be certified as secure and trustworthy.

Part I

Background, Related Work and Definitions

Chapter 2

Important Definitions

The core attack concept in this work is a hardware backdoor. In general computing and computer security, a *backdoor* is a term that refers to a method for bypassing normal or standard authentication, with the goal of gaining unauthorized access or executing unauthorized functions while remaining undetected by normal means. The main purpose of a backdoor, as opposed to other types of computer-oriented attacks, is to remain stealthy, which can allow for a long-term compromise of which defenders or security engineers remain unaware.

In this section, we present definitions that are necessary for the understanding of this work and how it expands on and contributes to computer security. These definitions are our own, and we hope that these definitions will allow for a more formal and standardized understanding of these subjects in the future.

Definition 1. *A* **hardware backdoor** *is any alteration to the circuitry in a computer chip that provides a method for intentionally violating the Instruction Set Architecture agreement.*

In other words, a hardware backdoor allows a piece of hardware to intentionally misbehave. We use the Instruction Set Architecture (ISA) as ground truth because the ISA is the contract that tells software what to expect from hardware. Any and all software that claims to be secure has as a core axiom the belief that the ISA is perfect. For example, if load and store operations do not behave as they are supposed to, it is impossible for any software program to trust its own memory operations.

We note that in some prior literature (mostly literature on foundry-side attacks), hardware backdoors are referred to as *trojans*. There is no tangible difference between backdoors and trojans, so for consistency we call all hardware-oriented attacks of this sort simply backdoors.

We next define the two properties that nearly all backdoors discussed in research literature have. Backdoors are usually *stealthy* and *digital*.

2.1 Digital Hardware Backdoors

We first define digital hardware backdoors and the alternatives to digital.

Definition 2. *A* **digital** *hardware backdoor is a backdoor for which all aspects of the backdoor's operation can be modeled using digital, cycle-accurate simulation.*

We note that this is a relatively strong definition, as it would be possible to use digital circuits to construct a backdoor that is not a digital backdoor. The reason research tends to focus on digital backdoors as we have defined them is because from both the attacker's and defender's point of view, backdoors can be well-understood if they can be simulated. Since many backdoors are implemented at the design level, using hardware description language (HDL) code, the ability to define backdoor functionality in terms of cycle-based, digital functionality is important.

Most research to-date has dealt with digital hardware backdoors that fit this definition, and most of the work in this thesis is also restricted to digital backdoors. However, some of the methods we discuss are generalizable to other types of backdoors. We define three types of non-digital backdoors that fill the rest of the space of possible hardware backdoors.

Definition 3. *An* **asynchronous** *hardware backdoor is a backdoor for which the behavior cannot be well-understood using only cycle-accurate simulation.*

There are several reasons a backdoor might be asynchronous. It might use properties of the clock, such as only functioning only during the negative clock edge or only if the clock jitter is sufficiently high. A backdoor might also be implemented on a system with multiple clock domains and might be reliant on a certain amount of clock misalignment. A backdoor

could also rely on naturally asynchronous components, such as an asynchronous reset signal or the clock generation logic itself. Any backdoor whose behavior is not well-defined in a discrete, clock-based model fits into this category.

Definition 4. *An* **analog** *hardware backdoor is a backdoor for which the behavior cannot be well-modeled with digital circuits.*

A backdoor can fit into this category by using any analog logic. This could be a backdoor implemented into an analog component, or it could be any other backdoor that makes use of analog logic in its functionality.

Definition 5. *A* **parametric** *hardware backdoor is a backdoor that cannot be modeled computationally due to its reliance on physical properties.*

Parametric hardware backdoors rely on physical properties in the environment to function. These could include environmental properties, such as temperature, voltage, etc. These could also include incidental properties, such as the amount of wear-and-tear on a transistor, the amount of noise in a physical fabrication process, or anything else that is based on physical properties. The defining characteristic of all of these backdoors is that the design itself does not explicitly express the backdoor through simulation, because the backdoor is only functional under restricted physical constraints.

2.2 Stealthy Hardware Backdoors

We next define stealthy hardware backdoors and the alternatives to stealthy backdoors.

Definition 6. *A* **stealthy** *hardware backdoor is a backdoor whose payload has less than a* $\frac{1}{n}$ *probability of activating during a given cycle of validation testing for a fixed value of* $n \in \mathcal{Z}^+$.

The purpose of a stealthy backdoor is to be dormant during validation testing so that the design under test passes validation and goes to market. For this reason, most academically discussed backdoors are stealthy, and the assumption in most literature is that backdoors applied in the real world must be stealthy. The value of n is parameterizable depending on

the assumptions of the threat model. Generally, n should be sufficiently large that there is no real danger of a validation engineer stumbling onto the backdoor payload. In practice, n often takes on the value of large powers of 2, such as 2^{32} or 2^{64}. A realistic attacker would be aware of the value of n in a given setting by knowing the basic validation procedures being used.

This definition is of course dependent on the choice of validation method. If validation test inputs are chosen uniformly at random, then the probability of a payload activating is simple to compute. If the validation test inputs are biased, we assume that an adversary knows the bias and can compute probabilities accordingly.

Definition 7. *A* **frequently active** *hardware backdoor is a backdoor that is often or always performing malicious actions. A frequently active backdoor has a probability of activating during a given cycle of validation testing of at least* $\frac{1}{n}$ *for a fixed value of* $n \in \mathcal{Z}^+$.

An attacker might choose to use a frequently active backdoor if the validation team is believed to be ineffective or if for some reason there is no validation team. Frequently active backdoors are easy to implement, as they are simply a consistent violation of the ISA, designed into a hardware module. Another reason an adversary might use a frequently active backdoor is if there is a systemic problem in validation. For example, if the validation team only checks the positive clock edge, a frequently active backdoor on the negative clock edge might go unnoticed.

Chapter 3

Background on Hardware Design, Fabrication and Security

We begin by discussing why trust comes into question in modern hardware design and what notions of trust we can and cannot assume. Trust is a fundamental component in the development of complex, modern hardware systems. Myriad factors conspire to make hardware more susceptible to malicious alterations and as a result less trustworthy than was the case in the not-so-distant past. These factors include the growing use of third-party intellectual property (IP) components in system-on-chip designs, the global scope of the chip design process, increased design and integration complexity and design teams with a relatively small number of designers responsible for each sub-component. There have already been unconfirmed reports of compromised hardware [Uni, 2005; Simonite, 2013] leading to serious economic consequences [len, 2006]. A non-technical solution to the problem is to design and manufacture hardware locally in a trusted facility with trusted personnel. This solution is not a long-term or viable solution, as it is neither efficient nor guaranteed to be secure. This is why hardware security has grown as an academic area of research.

To understand how hardware can be compromised, we need to understand how hardware is designed. Figure 3.1 provides a bird's eye view of the process. The first few steps are similar to software design and construction: it begins with the specification of design requirements. Then the hardware is designed to meet operational requirements and coded

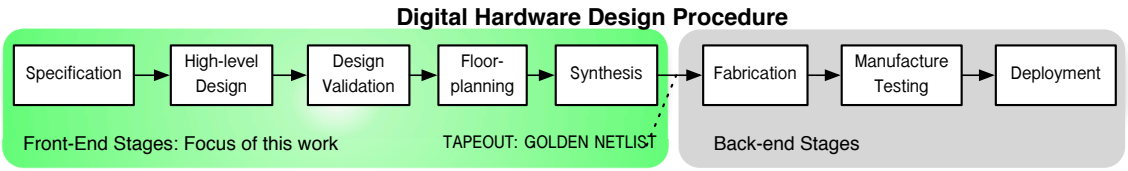

Figure 3.1: The front end and back end of the modern supply chain.

into a hardware design language (HDL), such as Verilog or VHDL. This coding can be done by designers working with the company, or the code can be purchased as intellectual property (IP) from third-party vendors around the world. This use of third-party IP is especially common in the case of peripheral components, such as USB controllers, which can be included within a larger hardware system. Hardware then undergoes validation testing. When compared against software systems, this testing can be much more involved. This is due in large part to the fact that, unlike software, hardware bugs are often extremely expensive or impossible to fix after deployment and can even result in product recalls. The reason for this difference is simply because software is easily mutable and can be patched, while hardware is for the most part immutable.

To minimize bugs, reputable hardware companies often employ validation and/or verification teams that are even larger than the design team. These validators either work in tandem with designers (if the designers are insiders) or after-the-fact (if the designs are purchased from third-party providers). The design, with all of its components, is then processed using computer-aided design (CAD) tools from commercial companies that convert the high-level code into gates and wires. This work can also be done manually in rare cases at the hands of custom circuit engineers. Once this process has been completed, there exists a functional design, which can be reviewed for security and/or reliability. Unfortunately, the state-of-practice is to simply send the design off to a foundry for manufacturing without any security enhancements. Code reviews are encumbered in large part by the complexity of the design and the pressures of time-to-market constraints.

Thousands of engineers may have access to hardware during its creation, and they are often spread across organizations, countries and/or continents. Each hardware production step has to be considered as a possible point of attack. Designers (either insiders or third-

party providers) may be malicious. Validation engineers may seek to undermine the process. CAD tools that may be applied for design synthesis prior to manufacture could be malicious. Finally, a malicious foundry could compromise security during the back-end manufacturing process. The primary root of trust is the front-end design phase, as without a trusted design (generally referred to as a 'golden' design) to send off to a foundry, we have no basis from which to begin to secure our foundries and the physical manufacturing processes therein.

When developing our trust model, we begin with one mild assumption that allows us to lessen our need for other trust assumptions.

3.1 Assumption of Trust at the Top

We make one key assumption of trust, not due to an optimistic perspective on trustworthiness but out of necessity. We believe that organizations that aim to make a profit or stay in business are unlikely to sabotage their own designs or sell damaged products intentionally. It is more likely that there are one or few 'bad apples' within an organization that are attempting to subvert the design or that the attack comes from external parties. In the unlikely case that an entirely malicious company sets out to produce and sell broken hardware, there is little one can do except wait for that company to go bankrupt or get sued. Similarly, in a military setting, if the entire military complex is malicious from the top down, there is not much that generic security techniques are going to do to help with that situation. We are interested in security methods that an organization, military or nation can use to protect itself and have trust in the hardware that it produces.

Given this mild trust assumption, we consider four major sources of insecurity concerning the necessary steps in the hardware design process. These sources of insecurity are 1) any third-party vendor from which IP is acquired, 2) any local or inside designers used in the design process, 3) the validation and verification team, and 4) malicious CAD tools or libraries.

All four of these potentially malicious agents are relevant, and we consider the possibility that one or all of them may be malicious, despite the fact that they may have been hired or dealt with in good faith by a benign organization. The most obvious and immediate

threats come from third-party vendors. This is because modern designs can contain tens to even hundreds of distinct third-party IP components, many of which are sourced from small groups of designers. These third-party components may meet functional specifications but may often do more than they are supposed to. Another significant threat is from rogue insider designers, such as a disgruntled employee or an implant from a spy agency. In addition to these individual conspiracies, we also consider the possibility of a larger conspiracy of malicious designers, including one between members of the validation and verification team within a company and third-party IP vendors from outside the company.

We trust a small number of security engineers (a few or potentially just one) to be trustworthy to use or implement our defensive techniques correctly, and we assume that untrusted personnel cannot go back and modify the design after the security techniques have been applied to the design. To keep this defensive effort small and thus make this assumption reasonable, we aim to produce low-overhead and simple security designs.

3.2 Related Work on Hardware Security

Hardware systems are increasingly becoming large ecosystems, consisting of many interconnected parts. There has been a significant amount of work over the past several decades on protecting different aspects of these systems. We discuss prior work on discovering and countering threats against a variety of different hardware components and systems.

3.2.1 Hardware Attacks that Pre-Date Backdoors

In addition to hardware backdoors, post-fabrication attacks are a wide area of attacks that can be perpetrated without compromising the hardware supply chain. Hardware, by which we mean collectively the processor, memory, network interface cards, and other peripheral and communication devices, is susceptible to two broad categories of attacks: 1) non-invasive side-channel attacks, and 2) invasive attacks through external untrusted interfaces/devices.

Physical side-channel attacks compromise systems by capturing information about program execution by analyzing physical emanations, such as electromagnetic radiation [Harada *et al.*, 1997; Mangard, 2003; Mulder *et al.*, 2005; Gandolfi *et al.*, 2001; Quisquater and

Samyde, 2001] or acoustic signals [Marchetti and Marks, 1974; Shamir and Tromer, ; Asonov and Agrawal, 2004], which occur naturally as a byproduct of the physics of computation. These attacks are an examples of covert channels [Lampson, 1973] and were initially used to launch attacks against cryptographic algorithms and artifacts (such as 'tamper-proof' smartcards [Mangard *et al.*, 2007][Kocher *et al.*, 1999]). General-purpose processors are also vulnerable to such attacks. There have been several attacks that exploit weaknesses in caches [Osvik *et al.*, ; Bernstein, 2005; Neve and Seifert, 2006; Neve *et al.*, 2006; Osvik *et al.*, 2005; Percival, ; Aciicmez, 2007; Bonneau and Mironov, 2006; Osvik *et al.*, ; Aciicmez *et al.*, 2007c] and branch prediction [Aciicmez *et al.*, 2007d; Aciicmez *et al.*, 2007b; Aciicmez *et al.*, 2007a]. Some countermeasures against these threats include self-destructing keys [IBM, ; Suh and Devadas, 2007; Gassend *et al.*, 2002; Yu and Devadas, 2010] and new circuit styles that consume the same operational power irrespective of input values [Tiri and Verbauwhede, 2005b; Saputra *et al.*, 2003; Kömmerling and Kuhn, 1999; Coron, 1999; Tiri and Verbauwhede, 2005a] and microarchitectural techniques [Tiri *et al.*, 2007; Brickell *et al.*, 2006; Tiri and Verbauwhede, 2006; Verbauwhede *et al.*, 2006; Agosta *et al.*, 2007].

Invasive device attacks are typically carried out by knowledgeable insiders who have physical access to devices. These insiders may be able to change the configuration of hardware to cause system malfunctions. Examples of such attacks include changing the boot ROM, RAM, Disk or other external devices to boot a compromised OS with software backdoors. Other examples include stealing cryptographic keys using unprotected JTAG ports [Altschuler and Zoppis, 2008; Rosenfeld and Karri, 2010]. A possible countermeasure is to store data in encrypted form in untrusted hardware entities. Since the 1980s, there has been significant work in this area [Smith, 2004]. Secure co-processors [IBM, ; Dyer *et al.*, 2001] and Trusted Platform Modules (TPMs) [tcg, 2007] have been used to secure boot processes. More recently, enabled by VLSI advances, researchers have proposed continuous protection of programs and on-chip methods for communication with memory and I/O integration [Lee *et al.*, 2004; Elbaz *et al.*, 2009].

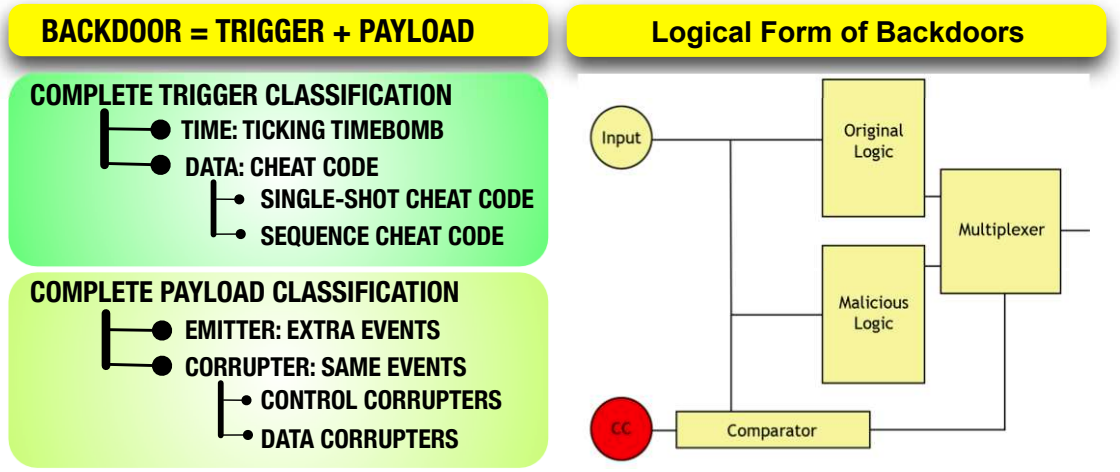

Figure 3.2: Taxonomy and illustration of digital hardware backdoors.

3.2.2 Hardware Backdoors

A more recent and more subversive threat is the threat of intentional backdoors, included at some point during the hardware development process. Broadly speaking, work in this area can fall into one of three categories: threats and countermeasures against malicious designers, threats and countermeasures against malicious design automation tools, and threats and countermeasures against malicious foundries. There has been work on detecting backdoors inserted by malicious foundries that typically rely on side-channel information such as power for detection [Wang *et al.*, 2008; Banga and Hsiao, 2008; Chakraborty *et al.*, 2008; Li and Lach, 2008; Banga *et al.*, 2008; Rad *et al.*, 2008; Salmani *et al.*, 2009].

There have been unconfirmed incidents of design-level hardware attacks [Adee, 2008] and work in academia on creating hardware backdoors. Shamir *et al.* [Biham *et al.*, 2008] demonstrated how to exploit bugs in the hardware implementation of instructions. King *et al.* [King *et al.*, 2008] proposed a malicious circuit that can be embedded inside a general-purpose CPU and can be leveraged by attack software to launch a variety of stealthy attacks. In just the past few years, the increased threat of hardware-oriented attacks has caused political and economic concerns [Simonite, 2013].

In one of the earliest works on protecting hardware designs from backdoors, Hicks *et al.* designed a method for statically analyzing RTL code for potential backdoors, tagging

suspicious circuits, and then detecting predicted malicious activity at runtime[Hicks *et al.*, 2010]. This hardware/software hybrid solution can work for some backdoors and even as a recovery mechanism. Its admitted weaknesses are that the software component is vulnerable to attack and additionally that the software emulator must itself run on some hardware, which can lead to infinite loops and DOS (denial of service).

Broadly speaking, a digital backdoors (in academia and likely in practice) tends to have the following logical form, illustrated in Figure 3.2. There is a good (functionally correct) circuit and a malicious circuit that exist together in a design. The outputs of both feed into a circuit that is semantically equivalent to a multiplexer (though it may be extremely hidden in terms of implementation). The multiplexer selects the output of the malicious circuit when a triggering circuit is activated. This logical form allows the circuit to behave exactly like a backdoor-free circuit whenever necessary, such as during validation testing.

3.2.3 Hardware Watermarking and Self-Identification

Hardware watermarking in general refers to methods for allowing hardware to have a unique identification or fingerprint. Watermarking is in essence a form of attestation, where the watermark attests to some property or simply the identity of circuitry. These methods have been discussed in academic literature as ways to help the foundry side of security. In surveying existing watermarking techniques, we break them down into heavyweight and lightweight solutions and discuss the strengths and weaknesses of existing solutions.

Heavyweight solutions allow for substantial overheads. They are often applied to large IP blocks in system-on-chip (SoC) designs. An example is constraint-based watermarking [Kahng *et al.*, 2001]. Constraint-based watermarks add unnecessary constraints to synthesis tools that are used to solve instances of NP-hard design problems, resulting in unique solutions to synthesis tasks. Other synthesis solutions — which an adversary might come up with — are likely to violate those constraints. The main limitation of this type of solution is that solving for these constraints is expensive, and imposing the constraints often hurts the quality and efficiency of the design [Abdel-hamid *et al.*, 2003].

Digital Signal Processing (DSP) watermarking involves using the partitions of a design to send out an identifying signal [Chapman and Durrani, 2000; Chapman *et al.*, 1999].

This method only applies to DSP components in SoC designs. However, it introduces the interesting notion that different pieces of a design can represent bits in a key that is not physically stored.

Lightweight solutions are often more practical but can have other drawbacks. An example is continuous side-channel watermarking [Becker *et al.*, 2010]. This method uses a small number of unnecessary gates to generate a small and continuous side-channel that can be measured. The drawbacks of this method are that it may be detectable by a malicious foundry, and it can be counterfeited if the foundry knows the technique is being used (since no obfuscation is used). Other lightweight solutions include finite state machine watermarking [Abdel-Hamid *et al.*, 2006] and test sequence watermarking.

3.2.4 Hardware Netlist Obfuscation

An alternative to watermarking that has been considered for foundry-side security is netlist obfuscation. The netlist, as the blueprint that is sent to the foundry, must be well-understood by an attacker before it can be tempered with. For that reason, obfuscating the (already difficult to interpret) netlist is a method that has been considered.

Obfuscation for hardware IP is a young area, and some of the initial solutions have been intended to raise the bar but not solve the problem altogether (in fact, a perfect solution is impossible). Two examples are comment removal and wire renaming, where comments and useful variable names are removed to make design code difficult to read. These types of methods are sometimes called layout obfuscation [Brzozowski and Yarmolik, 2007]. There also exist techniques for making local logic changes within a given module to make it harder to understand [Goering, 2003].

There exist further methods for obfuscating source code, such as breaking code unnecessarily into modules or reordering source code statements benignly [Brzozowski and Yarmolik, 2007]. These methods do not always change the resulting gatelists but do serve to make the source code harder to read. A similar proposal is to encrypt design IP source code and only allow the netlist to be viewed. Like other solutions of this type, this raises the bar against reverse engineering source code but does not prevent malicious foundries from reading the netlist (which they have to receive in unencrypted form in order to manufacture

the device).

Obfusflow [Chakraborty and Bhunia, 2008] (and related follow-up work [Chakraborty and Bhunia, 2009]) is a method for preventing IP piracy. The general idea is to embed an authorization code in the hardware that can be turned on to convert the hardware from obfuscated mode — wherein functionality is broken — to normal mode. In order to get correct operation, the taped-out chip has to match a stored (on-chip) sequence of codes. If that cannot be done, the behavior of the chip is essentially random, rendering it useless. The trade-off in this solution is area vs. anti-piracy (or anti-reverse-engineering) assurance because of the relatively high on-chip memory overheads.

Lastly, Koushanfar and Alkabani proposed a combination of watermarking and obfuscation for the protection of IP in sequential circuits [Koushanfar and Alkabani, 2010]. Their solution guarantees provable security properties though not true obfuscation in a theoretically secure sense (which again is impossible). The trade-off is that the overheads can get large, with the power overhead growing as large as 1,530% for the benchmark suite they considered.

Recently, key-based activation [Roy *et al.*, 2008; Rajendran *et al.*, 2012] has become more heavily studied. With key-based activation, one or more statically chosen keys are required to make a design function correctly. Without the keys, the design produces semi-random noise.

There has also been tangentially related work in garbling functions or circuits against functional evaluation (as opposed to inspection or reverse engineering) [Bellare *et al.*, 2012], as well as camouflaging gates against physical inspection [Rajendran *et al.*, 2013].

3.3 Perspective on Hardware Security as Compared to Software Security

We provide some perspective on how the field of hardware security in general relates to software security. The question we hope to answer in part is why it is that hardware security and software security operate in such different fashions.

3.3.1 Is Hardware Easier to Secure?

A natural question one might ask is whether or not hardware is easier to secure than software. In practice, hardware has less bugs than software (on average) and is less prone to attacks (so far). We believe that there is a fundamental aspect of hardware development that makes hardware easier to secure than software. The fundamental difference is that with software, attackers have an advantage, while with hardware, defenders have an advantage.

In the software setting, attackers have the tremendous advantage (in most settings) of having essentially unlimited time to develop and apply attacks. A piece of software is developed and then released into the world. Attackers can develop and try many different attacks against the essentially static piece of software. In other words, software is first developed and then attacked. Attacks have more flexibility and more time for mutation than software has. Thus, in the most general sense, one would expect that if an attacker and defender have equal resources, the attacker will win. In practice, this observation has shown itself to be true most of the time.

In the hardware setting, the game is very different and is almost the opposite. Since hardware does not normally change after it has been manufactured, it is the attacker who gets only one chance to achieve his or her goal. In fact, since the defenders (security engineers or testers) will usually get a chance to interact with the hardware after the last time an attacker has touched it, it is the defender who has the advantage in terms of time and flexibility. For this reason, one would expect that if an attacker and defender have equal resources, the defender should win. We believe this to be the case most of the time in the hardware setting. This is why we believe that a rigorous approach to hardware security can result in a satisfying solution that is not easily assailable even by sophisticated attackers.

3.3.2 Is the Economy of Hardware Different?

Another natural question one might ask is whether or not the economics of hardware development are fundamentally different from the economics of software development as they pertain to reliability and security. It is empirically the case that hardware has less bugs on average than software. We believe that there exist fundamental reasons for this.

The most important reason in our opinion has to do with timing. Software is often

developed with extreme time-to-market constraints and with the understanding that it can be patched at a later date. Thus, software is often released with known bugs to save time and money. Software is highly mutable and expected to be somewhat unreliable. As a result, that is the way software is generally built. In some secure settings, software is designed in a more rigorous fashion. However, this is less common. Additionally, even more security-aware software may be reliant on buggy software underneath, such as operating systems or hypervisors.

On the other hand, hardware development is economically motivated to be reliable. While in some cases it may be possible to 'patch' hardware, for example by redefining the ISA, bugs in hardware can result in complete product recalls or in forced downgrading of products (such as re-marketing an 8-core chip as a 4-core chip). While time-to-market constraints remain a factor, the cost of a recall is so high that it motivates companies to take some extra time to ensure reliability. As a result, hardware tends to have less bugs. Additionally, hardware development naturally has more time and more opportunity to include security mechanisms that might have non-negligible costs, especially if they can prevent catastrophic losses down the road.

Part II

Threat Model

Chapter 4

Threat Sources

When developing a threat model for a new space of attacks, it is important to consider the question: *from where can threats come?* In the case of hardware systems, threats can come from anywhere in the hardware development process.

As discussed in Part I, we believe that organizations that aim to make a profit or stay in business are unlikely to sabotage their own designs or sell damaged products intentionally. As a matter of definition, we do not consider it to be a hardware-oriented attack if an entire company knowingly produces a bad product. A hardware-oriented attack occurs when an adversary maliciously causes problems for a company that wants to produce operational hardware. By looking at the hardware development process, we can enumerate from where these adversaries can come.

The first step in the process is hardware design and coding. An adversary at this stage in the process can be any third-party vendor from which IP is acquired, or it can be any local or insider designer being used in the design process. These two types of adversaries are largely similar, with the main difference having to do with validation testing. In the case of third-party IP, validation tests might or might not be included and might or might not be trusted. On the other hand, with inside designers, it is very likely that there would be a corresponding validation team that is at least partially trusted.

The second step is synthesis, including design synthesis (into a gatelist) and physical synthesis (into a netlist). The trusted entity for this step is the suite of CAD tools being used and thus the company that produced the CAD tools. In practice, these tools tend to

be trusted *a priori*, due to there being very few tools and those tools being heavily used over a long period of time. However, we do not consider it necessary to trust these tools. In fact, the types of attacks that an automated tool might apply to a design are essentially identical to the types of attacks a designer could manually include. Therefore, nearly any method for securing the design against malicious designers also applies to synthesis tools. As we will discuss later, having a form of attestation built into the netlist further ensures that the synthesis tools are not adding their own malicious contributions.

The third step is the foundry. The foundry can potentially be malicious in its entirety. Broadly speaking, a foundry can have two different malicious goals. The first could be counterfeiting and the second could be backdoor inclusion. From a security standpoint, we are more interested in the latter of these two goals. However, counterfeiting is still of relevant interest. In fact, if an insecure version of a design already exists, then a foundry can 'alter' a newer more secure version by simply providing a counterfeit of the old version. This connection to counterfeiting further highlights the relevance of attestation.

The fourth and final step is post-fabrication testing. Currently, this testing serves to aid only in reliability. These tests mostly detect chips that are completely broken due to fabrication errors. However, this same step could be used for security purposes, as we propose in this work. If a security engineer is used for post-fabrication testing, that engineer has to be trusted, because otherwise simply lying suffices to compromise the system.

Chapter 5

Adversarial Game Model

We model our understanding of hardware-oriented threats and defenses as a game. This game serves to highlight the qualities of hardware-oriented security that make it distinct from software-oriented security. Most notably, the primary object of interest (the hardware) is relatively immutable. As such, it is incredibly important who the last entity to alter the hardware is. In this game, the attacker(s) and the defender(s) take turns altering the hardware. The first alteration is the original coding of the design components, and the last alteration is the final stage of post-fabrication testing. Since it is so crucial who makes the last alteration, we explicitly highlight this aspect of the model.

5.1 The Principle of Last Action

Conjecture 1. Last Action Conjecture. *We conjecture that given two entities, 1) a malicious, intelligent adversary, and 2) a well-intending security engineer, if the two entities are equal in resources and ability, then the entity who last touches the hardware prior to release will win.*

Specifically, this means the following. A defender, *i.e.,* a security engineer or team, has methods for securing and defending hardware. An intelligent, malicious adversary, who can impact both the design and fabrication processes, knows all defensive methods and has ways to subvert them. Developing new defenses will not prevent this adversary from developing new attacks. Our conjecture states that since both sides have ways of defeating

each other, and since hardware, once released, is a static, immutable object, whoever alters the hardware last prior to release wins the game.

From this (somewhat obvious) conjecture, we naturally derive what we call the Principle of Last Action.

Theorem 1. The Principle of Last Action. *A hardware device can be secured if and only if (one or more) trusted security engineers are the last entities to interact with the device prior to its release.*

This principle follows directly from our conjecture. We note that this final interaction with the hardware could involve altering the device or simply running an attestation protocol. Either way, if the device is secure after this last alteration, and if the adversary never has another opportunity to alter it, then the device is known to be secure due to the immutability of the hardware from that point forward.

Chapter 6

Taxonomy of Threats and Attackers

In this chapter, we break down further the sources of attacks and the capabilities they possess, as well as our assumptions regarding them.

6.1 Attacker Capabilities

Every step in the hardware development life cycle can potentially be attacked. We outline the possible points of attack and the capabilities of the attackers.

• **HDL Design Stage:** At this stage, an attacker could be anyone involved in the coding and design process of the hardware design modules. These attackers could be malcontents or compromised personnel, either at a third-party design house or internal to a hardware development corporation. Additionally, an entire third-party design house could be compromised. Broadly speaking, the malicious actions will either occur in secret by insider designers, or they will be intentionally included in third-party designs. In either case the malicious code exists within the soft design while it is undergoing validation testing. The malicious circuitry is included directly into the source code and remains there both during validation testing and during fabrication. If a backdoor is included in this fashion, then even an honest foundry will manufacture the backdoor. Additionally, state-of-the-art techniques for securing foundries will not catch the backdoor, because the final device will match the

'golden' design (which in this case is not sufficiently golden).

• **Validation Testing Stage:** During the validation testing stage, malicious attackers could be participants in validation testing or verification actions (if those are incorporated into the same process). We assume that it is not the case that the entire validation and verification team is conspiring maliciously. This is a necessary assumption, because if the entire validation team is malicious, then there is no notion of testing on which to ground security tests and inclusions (this is essentially the same as the assumption that there is not an entirely malicious organization that intentionally produces faulty chips. Thus, we assume that while validation tests may be faulty or incorrect, mistakes in validation are either accidental or hidden. Thus, while validation tests may fail to discover hidden backdoors, they will not ignore gross problems (such as a design that simply does not work at all).

• **Physical Synthesis and Layout:** In order to turn a soft design into a gatelist and then a netlist, which is the blueprint used by hardware manufacturers for physical fabrication, the design must be compiled, synthesized and laid out. This process is usually automatic but can also be done in a custom fashion by engineers. In either case, there is a trust issue, whether it is with regards to the custom circuit engineers or the software engineers who programmed the tools. We observe, however, that any backdoor that might be included during physical synthesis could have also been included in the HDL in the first place. Thus, this attack scenario is not dissimilar from attacks by malicious designers. In other words, the capabilities of automated code manipulation tools are the same as the capabilities of designers. The main difference exists in the way we apply the Principle of Last Action. While it is easy to include security circuits after the design and validation processes have completed, they may need to be applied prior to physical synthesis and layout. Thus, a malicious physical synthesis tool could remove the defensive circuits we added to the design. Granted, this would require extremely sophisticated software in the hands of the attacker, but it is worth considering. Therefore, we note that design-level security practices alone cannot protect against attacks from physical synthesis tools. For this reason, physical synthesis is more closely associated with fabrication processes. Only by using methods that also protect against malicious foundries or by applying defenses directly to the netlist can

we protect against malicious synthesis tools and malicious custom circuit designers.

• **Physical Fabrication:** Physical fabrication often occurs at foundries that are not only independent (third-party) but also international, and as such governments and large corporations may find it difficult to trust them. We consider the possibility that the entire foundry is malicious and can even conspire with malicious designers or other malicious agents. This is a strong threat model but not necessarily an unrealistic one. A malicious foundry can include its own hardware backdoors, similar to the ones that might be included by a malicious designer. Additionally, a foundry also has the power to simply apply counterfeits. For example, consider that security circuits have been applied to a design to secure it. If the foundry has the design for the original (unprotected) design, then that design can be fabricated instead of the one with defensive circuits. This is another key application of the Principle of Last Action. Since the foundry operates after all design-related aspects have been completed, design-level defenses alone can never be enough. Instead, we know from this threat model that the last action must always take place post-fabrication and must be performed by security engineers. We will see later how this Principle of Last Action motivates the use of post-fabrication attestation to ensure that security circuits exist even after the final step in the process.

We next discuss the assumptions we make with regards to hardware design and fabrication practices.

6.2 Assumptions Regarding Hardware Design Practices

Hardware designs can broadly speaking come from two types of sources, internal or external (third-party). We make the following assumptions about hardware design practices.

• *Assumption #1: Division of Labor* Typically, a hardware design team (such as a team designing a microprocessor) is organized into sub-teams, and each sub-team is responsible for a portion of the design (*e.g.,* fetch unit or load-store unit). Microprocessor design, as well as the design of other larger hardware systems, is a highly cooperative and structured activity with tens to hundreds of participants [Appenzeller, 1995]. The Intel Atom Processor, for instance, is reported to have had 205 "Functional Unit Blocks" [ana,]; a design of a recent

System-on-Chip product from ST Microelectronics is reported to have required over 200 engineers hierarchically organized into eight units [edn, 2008]. We assume that any sub-unit team in a design can be adversarial but that not all of the sub-units can be simultaneously compromised. Specifically, for a design process with $n > 1$ design teams, at most $k < n$ of them may be malicious. The design parameters of hardware defense systems depend on the expected value of k. While one could imagine powerful adversaries, such as adversarial nation-states, buying out complete teams to create undetectable malicious designs, it is more likely that attackers will be a small number of compromised personnel.

• *Assumption #2: Design Code Access* Malicious adversaries may be insiders, including malicious microprocessor designers, which include chip architects, microarchitects, RTL designers and verifiers, and circuit designers. These workers have approved access to the design, privilege to change the design, and an intricate knowledge of the microprocessor design process and its workings. A malicious designer will be able to provision for the backdoor either during the specification phase, *e.g.,* by allocating reserved bits for unnecessary functions, or by changing HDL code. We assume this will be unnoticed during the implementation phase and after the code reviews are complete. Our assumption that code audits will not be able to catch all hardware backdoors is justified because code audits in practice are not successful at catching all inadvertent, non-malicious design bugs.

• *Assumption #3: Extent of Malicious Alterations* A malicious designer is able to insert a backdoor: 1) using only a small number (tens or less) of bits of on-chip storage, 2) with a small number of logic gates, and 3) without cycle level re-pipelining. This assumption does not restrict the types of attacks allowed. However, we assume the attacker is clever enough to implement the changes in this way. This assumption ensures that the malicious designer can slip in the hardware backdoor unnoticed past traditional audit methods with high probability.

• *Assumption #4: ROMs* We assume that ROMs written during the microprocessor design phase contain correct data. In particular, we assume that microcoded information is correct. The reason for this assumption is that the data in ROMs is statically determined and not altered by the processor's state. For this reason, we consider this security issue to be better

solved statically than at runtime.

• *Assumption #5: Third-Party IP* We assume that designs acquired as third-party intellectual property (IP) do not include exhaustive validation suites. In practice, provided IP may come with some amount of included validation testing. However, this testing is unlikely to be complete or even trustworthy. Internal validation engineers may develop their own test suites for the design IP. However, these tests are also unlikely to be exhaustive in practice, due to feasibility constraints and/or time constraints.

6.3 Types of Digital Design-Level Backdoors

If a backdoor is implemented at the design level, then it fundamentally must contain two parts: a *trigger* and a *payload*. The payload is the malicious end achieved by the backdoor, such as the transfer of data or changing of a value or address. This payload causes some malicious action that has value to the attacker. The trigger allows the payload to occur at an appropriate time. Specifically, a trigger is a circuit that controls whether or not the payload is active at a given point in time (or clock cycle). The trigger is a fundamental component of a *design-level* backdoor, because including the backdoor prior to validation means that there must be a way to keep the payload dormant and unseen during validation tests. For a backdoor to be viable, the probability of its payload becoming active during validation tests must be either zero or negligibly small.[1]

6.3.1 Types of Backdoor Triggers

A design-level attacker can use two general strategies for triggering an attack: a time-based trigger or a data-based trigger. From the HDL perspective, input data and the passage of time are the only factors determining the state of the microprocessor, so these two strategies (or some combination of them) are the only ones possible.

• *Trigger #1: Cheat Codes* A malicious designer can use a sequence of uncommon bits, embedded in either the instruction or data stream to unlock/lock the backdoor. For instance, a store instruction to a specific address and a certain value (one pairing in a 2^{128} space for

[1] In practice we usually consider values on the order of 2^{-64} or smaller.

a 64-bit microprocessor) can be used as a key to unlock a backdoor. Since the search space is so large, the chance that this trigger is hit by random verification is negligible. King *et al.* describe a variant of this attack in which a sequence of instructions in a program unlocks a trigger. The cheat code method gives an attacker a very high degree of control on the backdoor but may require a reasonably sophisticated state machine to unlock the backdoor. Further, it requires execution of software that may not be possible due to access restrictions.

It is necessary to break down cheat codes into two subtypes, due to the existence of both time (clock cycles) and data in HDL. A purely data-based cheat code is a large data values that contains the full cheat code all at once. We call this a *single-shot cheat code.* Any other sort of cheat code must use a notion of the time (the passage of clock cycles). For example, a cheat code could come in multiple pieces, or the entire cheat code could be inferred from a sequence of events, such as loads and stores to a memory unit. We call this latter type of trigger a *sequence cheat code.* We note that there is a natural mapping of sequence cheat codes onto *control* interfaces and single-shot cheat codes onto *data* interfaces. This is because, in practice, data interfaces are large enough to handle single-shot cheat codes, while control interfaces interpret the sequences and types of operations occurring. These are not hard restrictions but can help pedagogically in understanding the differences between sequence cheat codes and single shot cheat codes.

• *Trigger #2: Ticking Timebombs* An attacker can build a circuit to turn on a backdoor after the machine has been powered on for a certain number of cycles. The timebomb method is very simple to implement in terms of hardware; for instance, a simple 40-bit counter that increments once per processor clock cycle can be used to open a backdoor after roughly 18 minutes of uptime at 1 GHz. Unlike the cheat code method, timebomb triggers do not require any special software to open the backdoor. Timebomb triggers can easily escape detection during design verification because random verification tests are typically not longer than millions of cycles.

6.3.2 Types of Backdoor Payloads

We classify the possible digital backdoor payloads into three categories, based on implementation characteristics at the microarchitectural level. The main point of focus is whether or not the payload generates additional microarchitectural transactions. The motivation for this perspective is the following. An adversary is inherently motivated to generate extra transactions as a payload. By doing so, the attacker can perform extra work without impacting architecturally visible state. An example of this could be a code injection payload, where a computer virus runs in between the issuing of one instruction and the next. This type of payload allows for simplicity of implementation and invisibility, which are two aspects that an attacker is likely to optimize for. On the other hand, an attack can develop a payload that does not inject transactions. In that case, the attack must be significantly more aware of the nature of the running program, because it must change transactions in flight without crashing the system (unless the goal is simply a denial-of-service attack).

Thus, we observe that an attacker can either create a hardware backdoor to do more (or less) work than the original, uncompromised design would, or the attacker can create a backdoor to do the same amount of work (but work that is different from that of an uncompromised unit). This is (trivially) a complete, binary classification, which we will elaborate on further.

• *Emitter Backdoors* We define an emitter backdoor to be a backdoor wherein the payload generates additional microarchitectural transactions. An example of an emitter backdoor in a memory unit is one that sends out loads or stores to a shadow address. When this type of attack is triggered, each memory instruction, upon accessing the cache subunit, sends out two or more microarchitectural transactions to downstream memory units in the hierarchy. Similar attacks can also be orchestrated for southbridge (I/O control hub) components, such as DMA and VGA controllers, or other third party IP, to exfiltrate confidential data to unauthorized locations.

• *Corrupter Backdoors* We define a corrupter backdoor to be a backdoor wherein the payload alters in-flight transactions without generating new ones. In this type of attack, the attacker changes the results of a microarchitectural operation without directly changing the number of microarchitectural transactions.

We mentioned that there are three total types of backdoor payloads. These three types include emitter payloads, as well as two distinct types of corrupter payloads. It is necessary to break down corrupter payloads into two types because of the fundamental difference between *control* and *data* signals within microarchitectural designs. We correspondingly define these two subcategories of corrupter payloads as control corrupters and data corrupters.

A *control corrupter backdoor* causes microarchitectural transactions to change somewhere else on-chip (at a later cycle) by corrupting the type or semantics of an instruction in flight. For example, if a decode unit translates a no op instruction into a store instruction, this will indirectly cause the cache management unit to do more work than it would in an untampered microprocessor. However, this change will not manifest itself until a later cycle. This is different from an emitter attack because the decode unit does not insert any new transactions directly; it decodes exactly the same number of instructions in the tampered and untampered case, but the value it outputs in the tampered case causes the cache unit to do more work a few cycles later.

Data corrupter backdoors alter only the data being used in microarchitectural transactions. Examples of this could include changing the value being written to a register file or changing the address on a store request. Data corrupter backdoors can be used to change program flow, for example by changing a value in a register, thus changing the result of a future 'branch-if-equal' instruction. However, each individual instruction will still do the same amount of work as it should. The extra work will not occur until the corrupt instruction has been committed. Thus each instruction in a vacuum will appear to be doing the correct amount of work.

6.3.3 Trade-Offs Regarding Backdoor Payloads From an Attacker's Point of View

From an attacker's point of view, emitter backdoors are easier to implement. Emitter attacks, such as shadow loads (extra loads that exfiltrate data), have very low area and logic requirements in practice. They also have the powerful property that a user may not see any symptoms of hardware emitters when using applications. This is because they can preserve the original instruction stream.

Corrupter attacks, on the other hand, are more complicated to design and harder to hide from the user. In these attacks, rather than simply emitting bogus instructions, the user's own instructions are altered to invoke the attack. Since the user's instructions are being altered, the attacker must have detailed knowledge of the applications being run in order to alter data without tipping off the user. If the execution of the backdoor were to cause the user's program to crash, this would violate the secrecy of the attack. Corrupters are hard to design because corrupting specific data requires at least partial decoding of user instructions and becomes harder as the size of datapaths scales. For example, a malicious decoder that turns no ops into shadow loads becomes harder to implement as instruction decode becomes more complex. In the case of multi-stage decoders, the backdoor itself may require latches and execute over multiple cycles. For contrast, an emitter attack on the load/store unit can send out shadow loads without using any extra internal state by simply sending out shadow loads whenever it is not busy.

To summarize, the best value per effort for an attacker is from ticking timebomb emitter attacks (a backdoor that uses an emitter payload and a ticking timebomb trigger). Such attacks can be implemented with very little logic, are not dependent on software or instruction sequences and can run to completion unnoticed by users. In our solutions, however, we will discuss strategies to cover all types of backdoors, including all trigger types and all payload types.

Now that we have covered assumptions and capabilities of the design-side, we next discuss assumptions and capabilities of the foundry side.

6.4 Assumptions Regarding Hardware Synthesis and Fabrication Practices

It is commonly assumed in hardware security fields that hardware synthesis tools are trusted. However, we do not make this assumption. The main reason for trusting synthesis tools currently is that there are only a few commonly used tools, and they are both old and widespread. Therefore, if a synthesis tool has been corrupted, it has been corrupted on a grand scale, though we note that a compromise to a compilation tool has the potential to

persist for a long time [Thompson, 1984]. While this assumption of trust might very well be valid, we find it unnecessary. If it is possible that a synthesis tool can automatically insert backdoor logic, we want to be aware of that possibility. We note, however, that allowing for this possibility does not radically change the situation for us, because we already handle both design-side and foundry-side attacks.

The difficulty with synthesis tool-oriented attacks is that they can look either like design-side or foundry-side attacks. By making malicious modifications at the gate level, a malicious tool that implement a backdoor that could have been inserted at design time. By making intentional mistakes in placement and routing, a synthesis tool can implement a backdoor that normally would have to be added during fabrication. However, in either of these cases, equivalent attacks could have been added by either a malicious designer or a malicious foundry. Thus by allowing for both malicious designers and a malicious foundry, we argue that we are implicitly allowing for a malicious synthesis tool chain.

While our model of design-side backdoor triggers, payloads and capabilities is original and expands understanding of the field, our model of foundry-side capabilities is relatively standard. A malicious foundry has essentially unlimited capabilities. While they receive a netlist that clearly specifies the device that should be produced, there is nothing in place that ensures the integrity of the final product. At the most, there is a small amount of post-fabrication testing to check that the device is capable of basic operation.

There are three ways in which a foundry can produce a device that fails to match the given netlist.

• **Backdoor Insertion:** A foundry can manually insert backdoor logic into the netlist and manufacture the backdoor into the device. Broadly speaking, there are two types of such backdoors. The first is a *self-contained* foundry-side backdoor, where a backdoor is inserted into a legitimately trustworthy design. The second type is a *conspiracy-oriented* foundry-side backdoor. In this case, a conspiring malicious designer already included part or all of a backdoor in the design. However, either the backdoor has no trigger or that trigger has been disabled by defensive mechanisms. The foundry can enable that trigger, either by supplying it directly through netlist alterations or by removing the defensive circuits from the netlist. The result is that a design-side backdoor that would have been thwarted is now

operational. In either of these two settings, the result is that a harmless design results in a device that contains a backdoor.

• **Counterfeiting:** Counterfeiting is a field that has already received significant study for reasons other than hardware security. A foundry can steal IP and/or sell chips on the black market by counterfeiting designs. In the context of hardware security, counterfeiting becomes even more of a problem. The ability to counterfeit means that a foundry has the ability to ignore design-side protections. For example, consider that there exists an insecure design $\mathcal{I}$. A trustworthy design organization goes to great efforts to secure $\mathcal{I}$ using state-of-the-art design-side security algorithms, resulting in the creation of a golden design $\mathcal{G}$. The netlist for $\mathcal{G}$ is sent to an untrustworthy foundry for manufacturing. Without needing to understand $\mathcal{G}$ or go through any extra effort, the foundry simply produces counterfeits of $\mathcal{I}$ and markets them as $\mathcal{G}$. The result is that all of the security effort that went into $\mathcal{G}$, no matter what it is, is completely negated. We note that this example emphasizes the fundamental importance of the Principle of Last Action. The simple act of counterfeiting undoes any and all design-side security unless there exists testing that occurs after fabrication.

• **Accidental Errors:** The third and final way in which errors can be introduced is accidentally. Modern fabrication techniques are extremely complex, and the yield rates for chips are significantly less than 100%. Therefore, many chips simply fail to work by accident. Simple post-fabrication tests catch most of these errors, as it is easy to notice when a chip is non-operational. These types of errors do not concern us significantly, as we are more concerned with chips that are correctly manufactured but contain well-hidden, intentional faults. Intentional faults are of course unlikely to be caught by simple post-fabrication tests.

Part III

Designing Trustworthy Hardware

Chapter 7

Protecting Hardware Designs Proactively with a Defense-in-Depth Approach

Our approach to defending against design-level hardware-oriented attacks is based on two fundamental goals: to be proactive, and to apply defense-in-depth.

7.1 Why Be Proactive?

We believe that it is vital to be proactive when protecting against hardware-oriented attacks. One of the main reasons for this is the time scale on which we are operating. A piece of hardware takes a long time (and significant resources) to develop. Once fabricated, that hardware might be sold and used for years or decades. Thus, we cannot wait for attacks and then react to them. Additionally, due to the amount of time available during the hardware development process, attackers are likely to be aware of all details of our defenses. Therefore, our defenses must be proactive in defending against all possible attacks and must also be aware of how attackers will react to knowledge of how our defensive systems work.

Thus, we consider it necessary that our defenses are *proactive*, *i.e.,* they protect against complete classes of attacks, rather than only known attacks. Additionally, it is necessary that our defenses are *self-aware*, *i.e.,* they continue to function even when attackers know

exactly when, where and how they will be implemented.

7.2 Why Use Defense-In-Depth?

Many areas of computer security result in arms races. An arms race is a situation where the attacker and defender are on roughly equal footing, and increasing sophisticated attacks have to be combated with increasingly sophisticated defenses. Arms races occur because the offense and defense are playing a fair game. It is always best to assume that attackers will apply the same amount of resources and intelligence as even the strongest defender. Thus, attempting to win a fair game results in an unending arms race, where both attacker and defender continue to improve their strategies.

Instead, our goal is to make the game unfair for the attacker. The two possible ways to do this are through making the game unwinnable for the attacker or through defense-in-depth. We do our best to make the game unwinnable for attackers by applying the Principle of Last Action. However, doing so relies on human agents, such as attestation engineers. As such, the game will never be truly unwinnable for attackers. Therefore, we employ defense-in-depth as a supplemental strategy.

Our logic is the following. Given that the game as designed is unwinnable for attackers, they can only win by violating one of the rules (*i.e.,* axioms) of the game. Such violations might include bribing a security engineer, discovering a new technology (such as more advanced reverse engineering capabilities) or otherwise undermining an axiom believed to be true. Such violations may be difficult or infeasible, but security researchers know from experience that they can occur. However, with n-way defense-in-depth (n independent security mechanisms), an attacker is required to *simultaneously* compromise the axioms of all n systems. While defense-in-depth can never make attacks completely impossible, the growth in difficulty as a function of n is very rapid, making it highly effective in practice.

For these reasons, we apply defense-in-depth, in addition to making the game as unfair as possible for attackers.

Chapter 8

Functional Analysis of Hardware Design Code

The first of the three stages of our defense-in-depth approach to hardware security is static analysis. This step occurs during design and code entry, before the hardware has been manufactured. The design being analyzed can either be a design that was developed internally by an organization's own design team or a design that was acquired from a third-party organization. As this first line of defense, we have a developed the first algorithm for performing static analysis to certify designs as backdoor free and have built a corresponding tool called **FANCI** (Functional Analysis for Nearly-unused Circuit Identification) [Waksman *et al.*, 2013b].

Recall from the earlier section on backdoor models that a backdoor is turned on when rare inputs called triggers are processed by the hardware. Since the trigger inputs are rare, the trigger processing circuit rarely influences the output of the hardware circuit: it only switches the output of the circuit from the good sub-circuit to the malicious sub-circuit when a trigger is received. If we can identify sub-portions of a circuit that rarely influence the output, then we can narrow down the set of sub-circuits that can be potentially malicious. We call this set of potentially malicious circuits *stealthy*.

Boolean functional analysis helps us identify sub-circuits that rarely influence the outputs. We quantitatively measure the degree of influence one wire in a circuit has on other using a new metric called *control value*: the control value of an input wire w_1 on an output wire w_2 quantifies how much the truth table representing the computation of w_2 is influenced by the column corresponding to w_1. FANCI detects stealthy sub-circuits by finding wires that have anomalous, low control values compared to other wires in the same design.

The **FANCI** algorithm to compute the control value of w_1 on w_2 is presented as Algorithm 1. The control value is a fraction between zero and one quantifying what portion of the rows in the truth table for w_2 are directly influenced by w_1. In step 3 of the algorithm, we do not actually construct the exponentially large truth table. We instead construct the corresponding boolean function. Since the sizes of truth tables grow exponentially, to scale FANCI, we approximate control values using a constant-sized subset of the rows in the truth table.

To take a simple example, suppose we have a wire w_2 that is dependent on an input wire w_1. Let w_2 have n other dependencies. From the set of possible values for those n

Algorithm 1 Compute Control Value

1: $count \leftarrow 0$
2: $c \leftarrow \text{Column}(w_1)$
3: $T \leftarrow \text{TruthTable}(w_2)$
4: **for all** Rows r in T **do**
5: $x_0 \leftarrow$ Value of w_2 for $c = 0$
6: $x_1 \leftarrow$ Value of w_2 for $c = 1$
7: **if** $x_0 \neq x_1$ **then**
8: $count{+}{+}$
9: **end if**
10: **end for**
11: $result \leftarrow \frac{count}{size(T)}$

wires (2^n), we choose a constant number, let us say for instance 10,000. Then for those 10,000 cases, we toggle w_1 to zero and then to one. For each of the 10,000 cases, we see if changing w_1 changes the value of w_2. If w_2 changes m times, then the approximate control value of w_1 on w_2 is $\frac{m}{10{,}000}$. Once we have computed all of the control values for a given wire (an output of some intermediate circuit), we have a vector of floating point values that we can combine to make a judgement about stealth. We have found that using simple aggregating metrics, such as the arithmetic mean and median, are effective for identifying stealthy wires. Other metrics may be possible and interesting in the future. The complete algorithm used by FANCI is summarized in Algorithm 2.

An example of how backdoor circuitry can stand out within a module is displayed in the histogram in Figure 8.1.

Due to the way FANCI works, the algorithm is guaranteed to flag any stealthy combinational logic. This means that it is impossible to trick FANCI through conventional means. FANCI also works in practice against sequential (state machine-based) backdoors. While the tool does not take state into account, state-based backdoors generally require combinational logic to *recognize* the trigger state, and FANCI is (in our experience) able to catch that combinational logic.

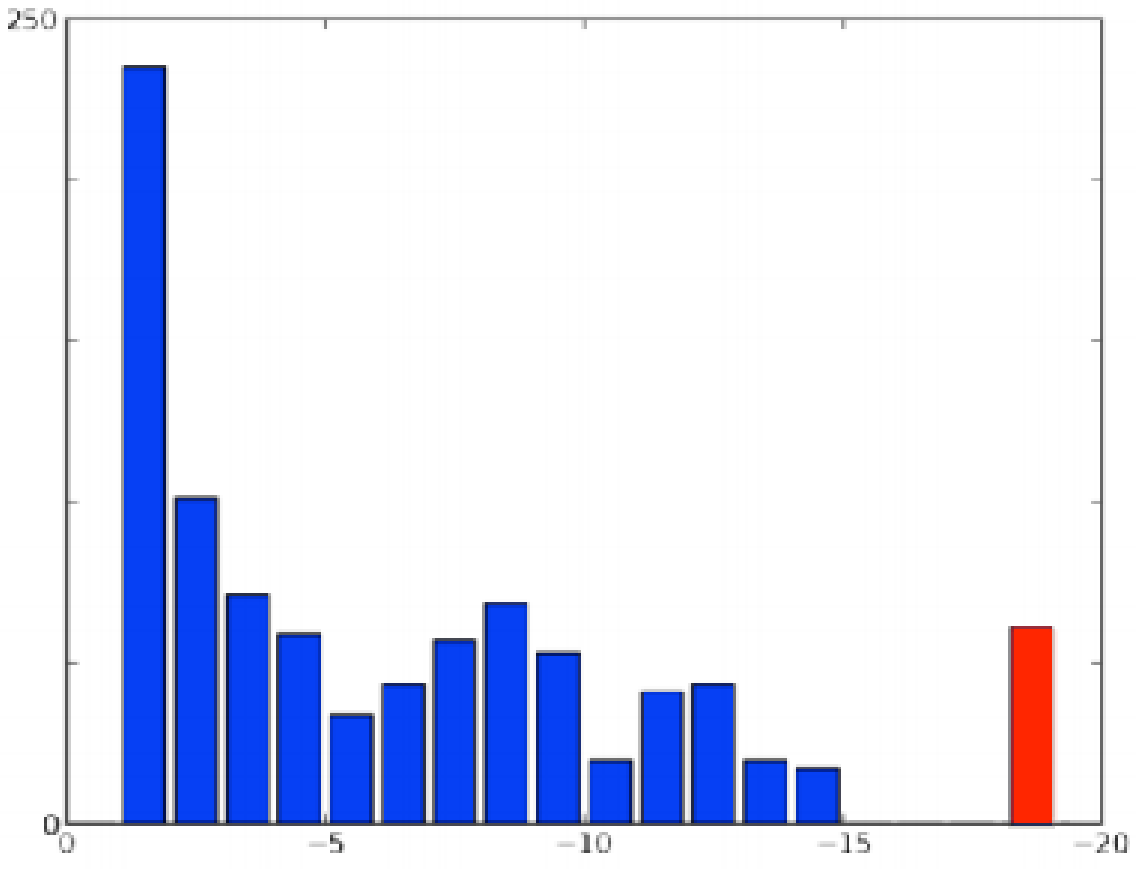

Figure 8.1: An example of how backdoor circuitry (in red) can stand out from normal circuitry (in blue) when stealth scores are calculated. Stealth values are shown on the X-axis on a logarithmic scale.

8.1 Heuristics for Identifying Stealthy Wires from Control Value Vectors

When we are done computing approximate control values for each input, we have a vector of values for each output in the design. Our control value measurements capture how one output wire is influenced by another input wire.

In this section we describe the heuristics that we use for making final decisions about wires in designs. Given a vector of control values, these heuristics take an average and determine whether or not a wire is suspicious. Having only one weakly-affecting wire or a

Algorithm 2 How FANCI Flag Suspicious Wires in a Design

```
1: for all modules m do
2:   for all gates g in m do
3:     for all output wires w of g do
4:       T ← TruthTable(FanInTree(w))
5:       V ← Empty vector of control values
6:       for all columns c in T do
7:         Compute control of c
8:         Add control(c) to vector V
9:       end for
10:      Compute heuristics for V
11:      Denote w as suspicious or not suspicious
12:    end for
13:  end for
14: end for
```

wire that is only borderline weakly-affecting might not be sufficiently suspicious. This is why we need heuristics for taking into account all of the control values in the vector.

Going back to the example where w_2 is our output, w_2 has a vector of $n+1$ control values from its inputs (w_1 and the n others), each between zero and one. These $n+1$ numbers are the $n+1$ control values from the dependencies of w_2. In this section, we discuss options for processing these vectors to make a final distinction between suspicious and non-suspicious output wires.

For a small but real example of what these vectors can look like, consider a standard, backdoor-free multiplexer with two selection bits that are used to select between four data inputs. This common circuit is depicted in Figure 8.3. The output M of the multiplexer is dependent on all four data inputs and both selection bits. Semantically, the selection bits choose which of the four data values is consumed.

We can see intuitively what the control values are for the six input wires (computation for one input is shown explicitly in Figure 8.3). The situation is symmetric for each of the four data wires (A, B, C and D). They directly control the output M in the cases when

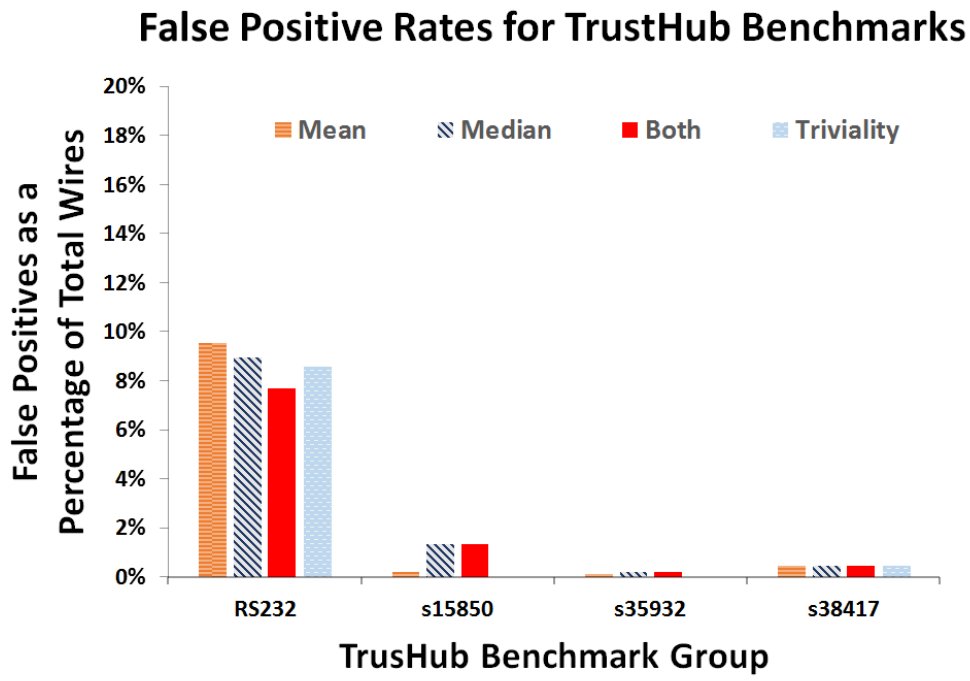

Figure 8.2: False positive rates for the four different metrics and for TrustHub benchmarks. The RS232 group — which is the smallest — has about 8% false positives. The others have much lower rates (less than 1%).

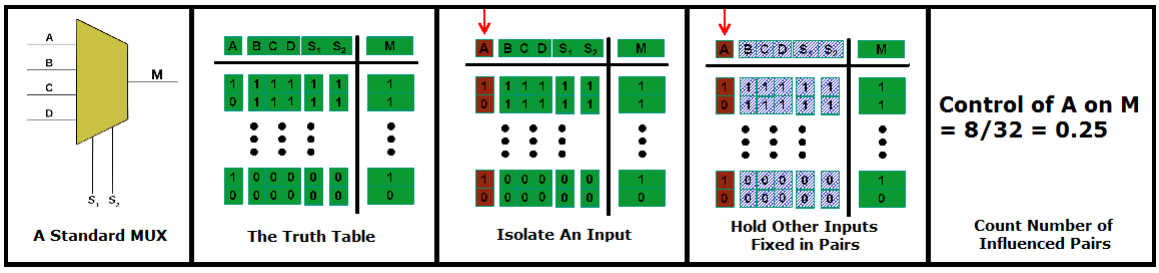

Figure 8.3: A standard 4-to-1 multiplexer. The output M takes on the value of one of the four data inputs (A, B, C, D) depending on the values of the two selection bits (S_1, S_2).

the selection bits are set appropriately. This occurs in one fourth of the cases, and each of these data inputs has control value 0.25. This can also be confirmed by writing out the truth table and counting the rows.

The two selection bits have higher control values. A given selection bit chooses between

Algorithm 3 Compute Approximate Control Value

```
numSamples ← N (usually 2^15)
n ← number of inputs
rowFraction ← numSamples / 2^n
count ← 0
c ← Column(w1)
T ← TruthTable(w2)
for all Rows r in T do
    if rand() < rowFraction then
        x0 ← Value of w2 for c = 0
        x1 ← Value of w2 for c = 1
        if x0 ≠ x1 then
            count++
        end if
    end if
end for
result ← count / numSamples
```

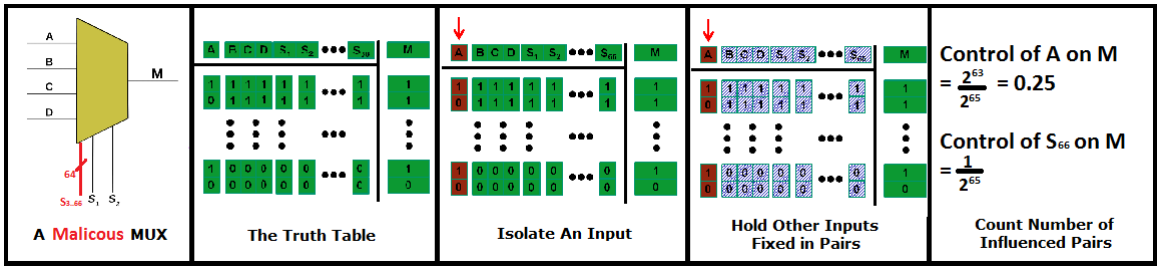

Figure 8.4: A malicious 4-to-1 multiplexer. The output M takes on the value of one of the four data inputs (A, B, C, D) depending on the values of the two selection bits (S_1, S_2). There are also 64 extra selection bits ($\{S_3, \cdots S_{66}\}$) that only change the output if they match a specific key.

two of the data values. For example, if $S_1 = 1$ then S_2 chooses between B and D. In that case S_2 matters if and only if $B \neq D$, which occurs in half of the cases. So the control

values for the two selection bits are 0.50. This can be confirmed by counting rows in the truth table.

The full vector of control values for the output M contains six values, one for each of the six inputs. The values are:

$$[0.25, 0.25, 0.25, 0.25, 0.50, 0.50]$$

Intuitively, this is a benign circuit, as we would expect. All of the inputs are in the middle of the spectrum (not close to zero and not close to one) which is indicative of a common and efficient circuit.

Figure 8.4 depicts a malicious version of multiplexer. In this case, there are 64 additional select bits. When those 64 bits match a specific 64-bit key, then the output of the mux is changed to a malicious payload. In terms of the truth table, this affects only an exponentially small fraction of the rows. The vector of values we would get for the output M would include 64 additional values for those 64 extra input wires. Each of those control values would be on the order of 2^{-63}. Intuitively, this is an obviously suspicious circuit. We next discuss heuristics for interpreting these vectors.

From a large circuit or large design, we get a variety of these control value vectors. The guarantee we have about the distribution of control values is at least one or a few of them will be zero or nearly zero for wires that belong to stealthy backdoor triggers. Thus, the vectors will contain at least some small values. The practical question is how to deal with these vectors and identify the output wires that are truly suspect from ones that are benign. Toward this end, we consider a few different heuristics for evaluating these vectors. The general description is shown in Algorithm 4.

Median: The first option we consider is the median. This is the median control value over all of the input dependencies that feed into a dependent output wire. This median value has the potential to be a good indicator of whether a set of dependencies is for the most part on the high end or the low end. In the case of backdoor triggers, the wires on the critical paths of the trigger generally have mostly unaffecting or very weakly-affecting dependencies. Thus, the median is often zero or very close to zero.

The median can be an imperfect metric when the data distribution is very irregular.

Algorithm 4 Compute a Heuristic for an Output Wire

```
1: w ← output wire
2: h ← heuristic function (e.g., median)
3: t ← threshold (between 0 and 1)
4: v(w) ← vector of control values
5: result(w) ← h(v(w))
6: if result(w) < t then
7:     return suspicious
8: else {result(w) ≥ t}
9:     return not suspicious
10: end if
```

This does not happen often but can happen. The reason for this is that the median does not reflect the presence of a few outliers in a large set of dependencies. Using just the median (as we confirm in our evaluation), can result in a few unnecessary false positives.

Mean: In addition to the median, we also consider the mean of the control values. Intuitively, the mean is an effective metric in the case of uniformly low dependencies or in the case where there is a heavy tail in the distribution. The mean is also more sensitive to outliers. For example if there are only a few dependencies, and one of them is unaffecting, that is likely to get noticed. However, one reason the mean might not be the best metric is because of this same sensitivity to outliers. It might fail to differentiate between two distributions where one contains more unaffecting wires than the other.

Both: Since there are potential limitations with both median and mean, we also consider the option of using both, *i.e.* flagging wires that have extreme values for both the mean and the median. We set a threshold for both the median and the mean, and we flag a wire as suspicious only if the median value is low and the mean value is also low. This helps in some cases to slightly diminish false positives. Details and comparisons are presented in Section 9.3.2.

Triviality: One last heuristic we consider in our implementation is one that we call *triviality*. This is a weighted average of the values in the control value vector. We weight them by how often they are the only wire influencing the output to determine how much

an output is influenced overall by its inputs. Equivalently, this heuristic computes the fraction of the rows in the truth table in which the dependent output wire has the value zero (or symmetrically, the value one). In practice we compute triviality directly in this way by looking only at the output column. The name 'triviality' refers to the fact that if the triviality value is zero or one then the circuit is completely trivial (always outputs zero or always outputs one). The exact value for triviality can vary from run to run depending on which rows are randomly selected, but it is probabilistically likely to vary by only a very small amount. Empirically, we did not see significant variance.

Many other heuristics could be considered in the future to attempt to gain incremental improvements in terms of false positive rates.

8.2 Relationship Between FANCI and the State-of-the-Art in Unused Circuit Identification

Prior to our work, Unused Circuit Identification (UCI) [Hicks *et al.*, 2010] was the state-of-the-art in analyzing backdoors inserted during the design phase. The state-of-the-art in design backdoor attacks is a class of attacks known as *stealthy, malicious circuits* [Sturton *et al.*, 2011]. This class of attacks deterministically evades UCI and was a viable way to attack hardware designs prior to our work. As we will see, **FANCI** catches stealthy, malicious circuits with extremely high probability.

UCI is an analysis algorithm that looks at dataflow dependencies in hardware designs and looks for completely unused intermediate logic. It is a form of dynamic validation, and in terms of our terminology, they identify always-affecting dependencies given a test suite. Given the inputs in the test suite, if two wires always carry the same value, there is an identity relationship, and the internal logic is unneeded. If the test suites were exhaustive, then UCI would have significantly fewer false positives. However, given the incompleteness of standard validation test suites, UCI has many false positives. For this reason, the Bluechip system was built to replace the removed logic with exception handlers that invoke runtime software in the case of false positives.

There are a few key differences between **FANCI** and UCI. The first is that **FANCI** does not require a validation test suite. This is valuable for two reasons. Today, third-party IP blocks often do not come with a validation test suite. Furthermore, if a validation suite is supplied, the malicious provider can change the validation test suite to help the compromised hardware evade UCI. A common problem in validation and verification is that achieving good code coverage and good interface coverage does not mean good coverage of internal states. Certain rare states may never get tested at all, which can lead to bugs in commercial designs and also offers ways for backdoor designers to evade detection. **FANCI** tests all logic equally, regardless of whether or not it is an input interface, and so it is impossible for a portion of the logic to go untested.

The second key difference between UCI and **FANCI** is that UCI is deterministic and discrete-valued in its approach. Given a test suite, a wire is only flagged if it is completely

unused, regardless of its relations to other wires. In **FANCI** we also catch nearly-unused wires, meaning wires that are not completely unused by which rare alter output signals. For example, if a wire affects the value of a nearby wire but ultimately has little impact on an output wire a few hops away, we will catch that. Another aspect of **FANCI** is that it takes into account the full vector of dependencies and uses heuristics to make a final decision. In all of the designs we tested, there were many always-affecting dependency relationships that **FANCI** did not flag. All of those relationships would have been false positives in UCI.

To give a toy example, consider a double-inverter path, two inverters placed one after the other. This is a logical identify function, so it generates an always-affecting relationship that would be flagged by UCI. However, as long as the output of the double-inverter path is used, it would not be flagged by any of **FANCI**'s current heuristics. This is a small example and could easily be hard-coded for in a practical implementation of UCI. However, it serves as a microcosm of the difference between the deterministic approach of UCI and the heuristic-based approach of **FANCI**.

Sturton *et al.* introduced stealthy, malicious circuits (SMCs) as a way to evade UCI. **FANCI** detects SMCs, and we explain the intuition behind why that is. The basic idea behind SMCs is to use logic that alters the values of intermediate wires but ultimately does not affect outputs. Roughly speaking, UCI looks for completely unused wires, while Sturton et al.'s class of attacks makes use of nearly-unused wires. Using this backdoor class, Sturton et al. demonstrate basic circuit building blocks — such as AND and OR gates — that can be used to implement stealthy hardware backdoors. Thus, any small backdoor can be turned into an SMC and evade UCI. The truth table for one of the simplest SMCs is the following (reproduced from [Sturton *et al.*, 2011]):

t_1	t_0	i_1	i_0	h	f	Operation
0	0	0	0	0	0	Normal Operation
0	0	0	1	1	0	Normal Operation
0	0	1	0	0	0	Normal Operation
0	0	1	1	1	1	Normal Operation
0	1	0	0	0	0	Normal Operation
0	1	0	1	1	0	Normal Operation
0	1	1	0	0	0	Normal Operation
0	1	1	1	1	1	Normal Operation
1	0	0	0	0	0	Normal Operation
1	0	0	1	1	0	Normal Operation
1	0	1	0	0	0	Normal Operation
1	0	1	1	1	1	Normal Operation
1	1	0	0	1	1	Malicious Operation
1	1	0	1	1	0	Malicious Operation
1	1	1	0	1	1	Malicious Operation
1	1	1	1	1	1	Malicious Operation

There are two normal input bits i_1 and i_0 and two trigger bits t_1 and t_0. In terms of the output f, this is a classic backdoor trigger. Only when all of the trigger bits are set to one does the functionality change. In the other cases, the functionality is fixed, and the circuit looks like f is the AND of i_1 and i_0. The use of the intermediate variable h, which is distinct from f, makes it so that t_1 and t_0 are not truly quiescent. Thus, Sturton proved that UCI's defenses could be evaded.

Can **FANCI** *detect stealthy, malicious circuits?* Observe that the trigger wires – t_1 and t_0 – are weakly-affecting for the output f, *i.e.,* they only affect the value of f during malicious operation, which is a smaller fraction compared to normal operation. This fraction diminishes as the trigger bits get numerous. Thus for the backdoors in this class of stealthy, malicious circuits, the trigger inputs will have low control values and will be caught by **FANCI** with probability approaching 1.

8.2.1 High-Level Understanding of Why FANCI Goes Beyond UCI

Philosophically, **FANCI** and UCI take similar approaches. They both leverage the observation that unused logic is potentially malicious. While the implementations are very different, at a high level, the core difference comes from the logic of how the notion of *unused* is formulated.

The notion of unused in the UCI setting can be expressed roughly as the following: *If* for a given path, in *all* test cases, that path is *always* unused, then the path is malicious. Otherwise it is not.

The notion of unused in the **FANCI** setting can be expressed roughly as the following: *If* for a given wire, *there exists* another wire that it is connected to for which in *most* possible cases, the path between the two wires is not used, then the wire *might* be malicious. Otherwise, it is probably not but still might be.

There are several differences worthy of note between these two formulations. Firstly, everything in UCI is deterministic, while everything in **FANCI** is probabilistic. The deterministic nature of UCI is what allows the attacks mentioned above to be guaranteed to work. The use of uniformly random row selection in **FANCI** makes it impossible for an attacker to deterministically avoid test cases.

Secondly, UCI uses all test cases, while **FANCI** uses a fraction of all possible cases, chosen at random. Thus, **FANCI** is not subject to the need for test cases and is also less predictable for an attacker.

Thirdly, while UCI looks at paths, **FANCI** looks at wires and the relationships to all other wires. For example, a path might do actual work and produce a useful value. That value might frequently affect one output. However, there might be a second output which is almost never affected by that value. Thus, while the path is being used to do work, there is a fraction of that work that is stealthy, even though it is not the case that the whole path is unused. This is similar to what happens in the 'stealthy, malicious' backdoors mentioned above.

Overall, UCI takes a deterministic, efficient and localized view in searching for unused circuitry. **FANCI** takes a less efficient, randomized and global view. In looking at all pairings of wires, **FANCI** does more work and potentially requires additional computation.

However, with the global view and the non-deterministic approach, it allows for wider coverage and does not expose itself to deterministic attacks.

8.3 Relationship to Boolean Function Theory and Fault Simulation

While the notion of *control* that underlies the algorithms used by FANCI arose naturally from the way hardware is designed, it is worth noting that similar concepts exist abstractly in the areas of boolean function theory and fault simulation.

8.3.1 Relationship to Shannon Cofactors

The intuitive notion of *control value*, used in the development of the algorithms that underly FANCI, is tightly connected with the mathematics behind *Shannon cofactors*. A Shannon cofactor is the commonly used term (especially in fault simulation and modeling literature) for the two 'halves' of a boolean function. Specifically, if a boolean function $\mathcal{F}$ contains a variable x, then that function has two Shannon cofactors with respect to x. The positive Shannon cofactor, usually denoted $\mathcal{F}_x$, refers to the function when x is fixed to the boolean value 0. The negative Shannon cofactor, usually denoted $\mathcal{F}'_x$, refers to the function when x is fixed to the boolean value 1. If the boolean function is thought of as its equivalent truth table, these two cofactors are literally the two halves of the table.

In the case of FANCI, we look at the difference between these two halves of the truth table when measuring control value. This bitwise difference is sometimes referred to as the *boolean difference* between two functions. The boolean difference can be computed between any two boolean functions that operate on the same input space, and in the case of FANCI we are looking at the boolean difference between the two corresponding Shannon cofactors of the boolean function representing a circuit. It would not be wrong to alternatively define the control value of an input wire on an output wire as the boolean difference between the two Shannon cofactors of the boolean function defining the output wire with respect to the input wire.

8.4 Evaluation of FANCI on Hardware Backdoor Benchmarks

For our implementation of FANCI, we developed a parser for gatelists that are compiled from the Verilog HDL, which is a popular language for the design of hardware. All of the concepts and algorithms we apply could be equivalently applied to VHDL or any other common HDL, as well as to hand-written gatelists. Our analysis is language agnostic, but we built a parser for structural Verilog and use Verilog for all evaluation purposes.

We used benchmarks from the TrustHub suite, which is the major academic benchmark suite for work on hardware backdoors [Tehranipoor *et al.*, 2012]. TrustHub is a benchmark suite from an online community of hardware security researchers and includes a variety of different types of backdoors, intended to be state-of-the-art in terms of stealth and effectiveness. For some of these benchmarks, the gatelists were provided. For others, we acquired the gatelists from the Verilog source using the Synopsys tool chain.

From the gatelist, our goal is to construct a representation of circuits that can be used to calculate different types of dependencies. The first step is to pre-process the gatelist to tokenize all of the different types and keywords. Each statement in a gatelist is either a declaration of wires, an assignment action, or the definition of a gate or small set of gates.

We treat all multiple-bit wires as sets of independent wires. It is the same for our purposes as if each bit was declared as its own wire. Gates that represent multiple basic logic functions — such as an AND-OR-INVERTER (AOI) — are broken down into their basic elements to make analysis easier. We treat memory elements (*e.g.,* flip-flops) as their logical equivalents. For example, a D-flip-flop is treated as an identity function. Basically, we are allowing them to be treated as pass-through gates. Since we analyze the internal logic of a module, as opposed to only the input and output interfaces, we are still able to catch sequential backdoors by analyzing the combinational logic that determines state transitions. Of course, doing exhaustive state analysis would be computationally intractable, which is why we do not pursue that direction.

We evaluate the four basic heuristics we presented for the TrustHub benchmarks. These are the mean, the medium, triviality and also the conjunction of median and mean. We

perform one run on each design[1] with $2^{15} = 32,768$ input cases (truth table row pairs), with the inputs chosen uniformly at random.

The most important result of our experiments is that we did not encounter any false negatives for any of the four heuristics. For every benchmark and for each of the heuristics, we discovered at least one suspicious wire from each backdoor, which was enough for us to identify the functionality of the hidden backdoors. Interestingly, different metrics tend to highlight different parts of the backdoor. In general, the mean and median tend to highlight backdoor payload wires and are similar to each other. This is because these payloads have triggers or resulting values from triggers as their inputs. Thus, several of the input wires have extremely low control values, causing both the mean and median to be small. On the other hand, triviality focuses more on the output wire itself and as such tends to highlight backdoor trigger wires. Since these are wires that are nearly always not set and only in rare circumstances get set (or *vice versa*), their truth tables tend to score very low for triviality. For each backdoor, we were able to identify at least one trigger and one payload wire. Using both metrics in concert can help out in code review by flagging more of the wires associated with the backdoor and thus demarcating the boundary of the backdoor more clearly.

Figure 8.2 shows the results for the 18 TrustHub benchmarks we analyzed with regards to false positives. For our results, we categorize the benchmarks into groups as they have been categorized by TrustHub. These categories represent four different design types, chosen to be part of the benchmark suite and containing a variety of backdoor triggering mechanisms. Each of the four groups contains a variety of backdoors manually included into a given design.

The RS232 group contains eleven benchmarks, representing eleven different backdoors applied to a relatively small third-party UART controller. The S35932 and S38417 groups each contain three benchmarks, containing backdoors built into two gatelists whose source and desciption are not provided. The S15850 group contains only one benchmark, also a gatelist without source or functional description. The S38417 group contains the largest designs in terms of area and number of gates.

[1] If desirable, multiple runs could be performed to increase confidence. In practice, the same results tend to come up every time, but it cannot hurt.

The RS232 benchmarks, as the smallest, mostly contain sequential (state-machine based) triggers but also a few combinational triggers. The s15850, s35932, and s38417 categories are more different from RS232 and more similar to each other. They contain mostly but not all combinational triggers and are significantly larger. We experienced a decrease in false positive percentage for larger designs. We attribute this to the fact that the total number of false positives remained roughly constant with respect to design size.

Additionally, the different benchmark categories achieve differing degrees of stealth. Most of the triggers in the RS232 category have a relatively high probability of going off randomly (such as during validation), as high as around one in a million. In the other categories, the probabilities are lower, ranging from one in several million to as low as around one in 2^{150}. The backdoors in the three low probability groups are the most realistic, since they are stealthy enough to evade detection by normal methods. The backdoors in the RS232 category go off with such high probability that validation testing would have a good chance of finding them. This is an aspect that made them more difficult to distinguish and resulted in slightly more false positives. From what we have empirically observed, the larger the design and the more well-hidden the backdoor, the better FANCI performs in terms of keeping false positive rates low.

False positives mean increased effort for security engineers. Once a wire is flagged as suspicious, a security engineer needs to look at the code and see if there is a good reason for that circuit to be there. If, for example, there are 1% false positives, that would mean roughly 1% of the design requires detailed code review.

Figure 8.2 shows the breakdown of false positives for different heuristics. Unsurprisingly, using the median by itself produced the most false positives on average. However, the difference is not large. The heuristic that produced the least false positives was triviality, again by only a slight margin. All four metrics are effective enough for practical use. We also believe that other metrics could be considered in the future to achieve incremental improvements in terms of false positive rates. A promising result we discovered was that the number of false positives diminished as a percentage when we looked at larger designs. In other words, scaling up to larger designs does not seem to greatly increase the total number of false positives.

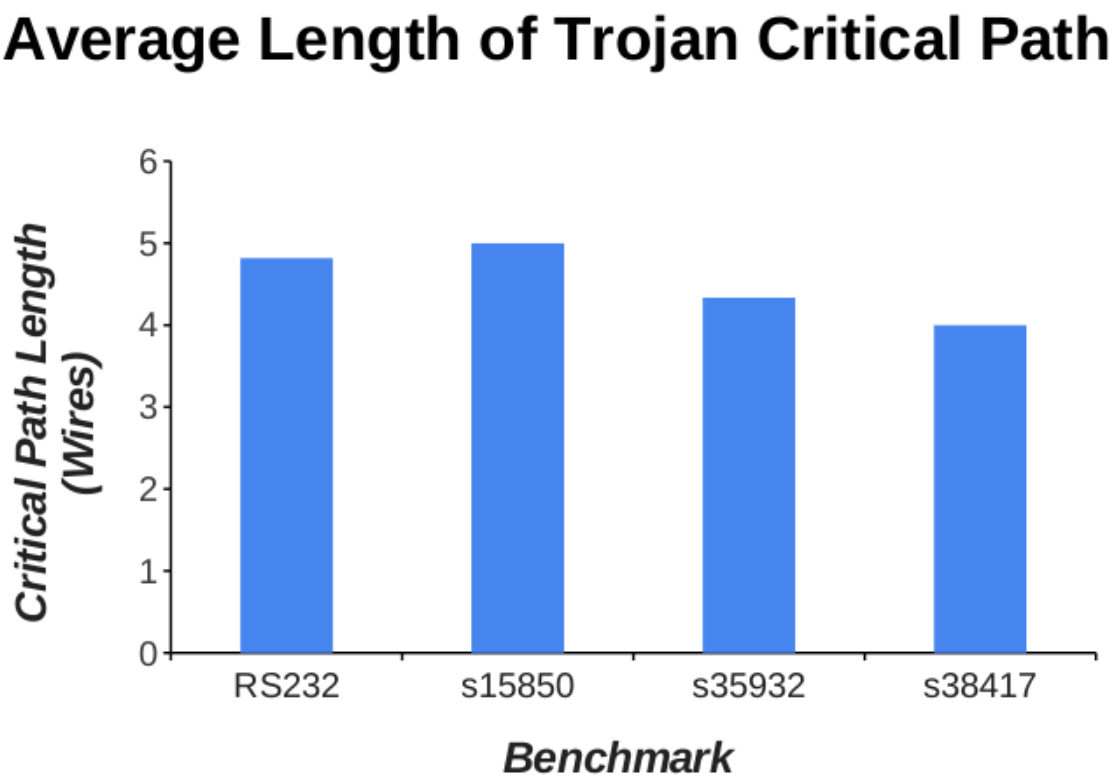

Figure 8.5: The average length of the backdoor trigger path identified in TrustHub benchmarks. Longer trigger paths are likely to be harder to detect because the distributions become more complex. The length is specified in number of distinct wires.

Figure 8.6 shows how many wires are flagged as suspicious on average for each of the benchmark groups by each of the different metrics. Each of the four metrics worked well, but triviality worked slightly better than the others. We see that all four metrics flag only a small number of critical wires, which means security engineers are given a small and targeted set to inspect. For most of the groups, FANCI whitelists more than 99% of the designs, making code review and inspection a feasible and relatively painless task. Triviality returns slightly fewer on average due to having slightly fewer false positives. It is not clear whether or not this difference is large enough to be statistically significant.

We next look at the size (using path length as a proxy) of the backdoors we encounter. Figure 8.5 shows the lengths of the backdoor trigger computations in the TrustHub benchmarks. The length here is specified in terms of number of distinct wires (or equivalently

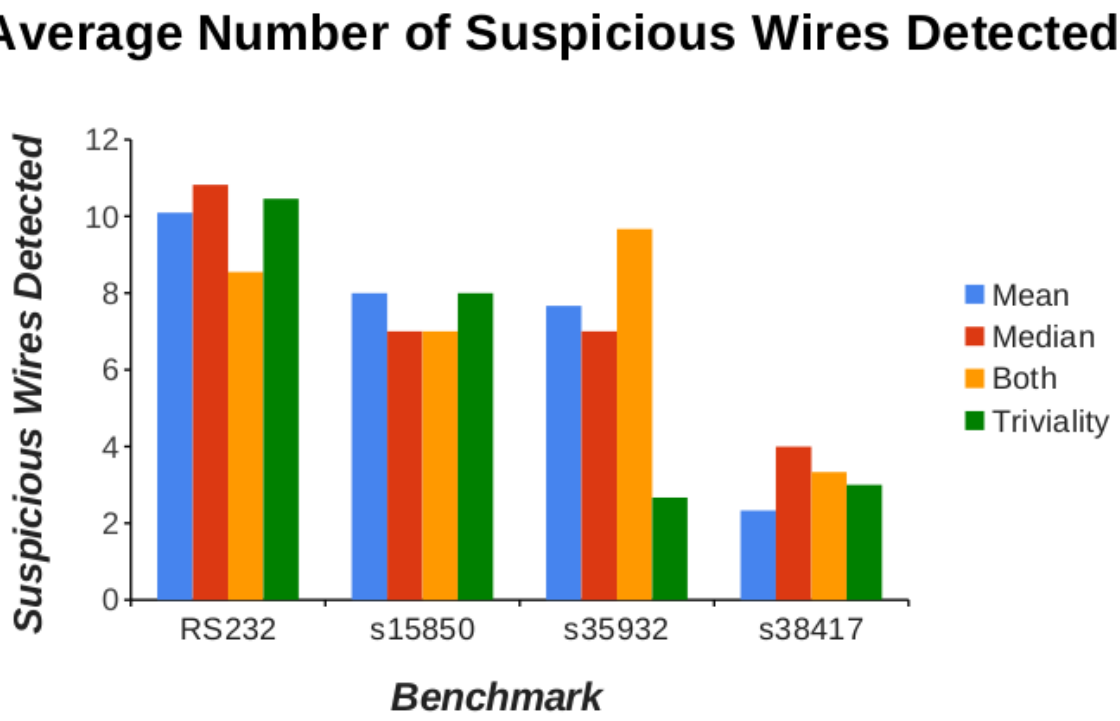

Figure 8.6: These are the total number of suspicious wires detected by each method for each type of backdoor design on average. For each design and each of the four methods we tried, we always found at least one suspicious wire. Thus, each of the four methods is empirically effective. However, some turned up larger portions of the trigger critical paths, proving to be more thorough for these cases.

number of gates). This is the way to think of lengths that makes the most sense for us from the perspective of digital boolean logic. Qualitatively, the smallness of these backdoors helps FANCI to work well. Since backdoor triggers need to be small and stealthy but also produce rare behavior, designers are forced to make each input wire as weakly-affecting as possible. These results could be commentary on the types of backdoors those designers choose to build, or it could be representative of the way people design malicious circuits in general. Without a wider array of benchmarks, we cannot say for certain. However, it appears that the crucial part of a backdoor – even a relatively complex backdoor – tends to be composed of only a few wires. This is good for security engineers because it means that we have a clear target for identification.

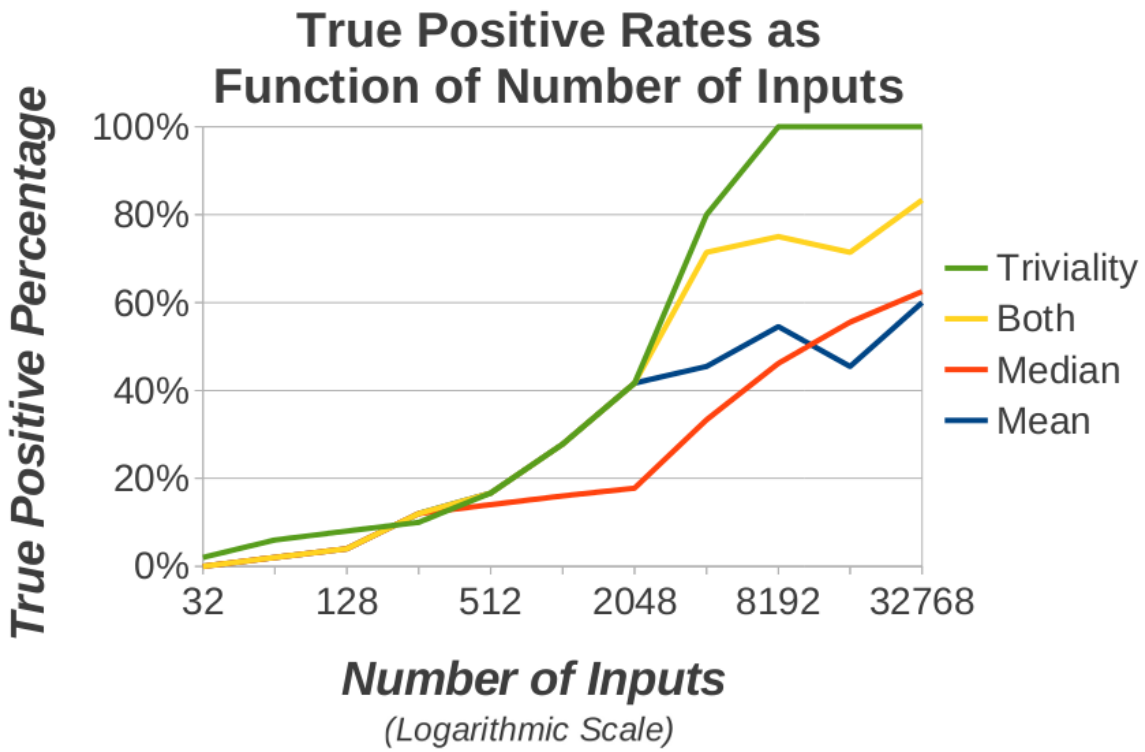

Figure 8.7: The trade-off between number of inputs testing (*i.e.* running time) and the true positive rate. Results are shown for four different metrics. The x-axis is on a logarithmic scale, so it takes a lot of inputs to achieve the best results. Running 32,768 inputs through a design generally takes between a few minutes and an hour.

We lastly test to see what happens as we increase and decrease the number of input rows we sample. The results for one of the benchmarks is shown in Figure 8.7. We see that up to a certain point, the results improve; after that point, the results tend to converge and stay about the same. This is essentially the law of large numbers kicking in, and it allows FANCI to scale well.

What we also learn from Figure 8.7 is that there are two sources of false positives. The first source is approximation. If we run only a few inputs, we get a lot of false positives, and if we run more inputs we get less false positives. The second source is from unavoidable false positives, i.e. weakly-affecting signals that are in the design for legitimate reasons. As Figure 8.7 shows, the true positive percentage plateaus quickly, which is why false positives

due to approximation are not a major concern.

The runtime for FANCI is roughly proportional to the size of the design under test in terms of number of total gates. In practice, the runtime ranges from a few minutes to about an hour using 2^{15} rows per truth table. The runtime can be increased or decreased by changing the number of inputs tested. In practice we did not find it necessary to decrease this number. Given the sizes of third-party IP components on the market, the runtime for FANCI should not be a problem for real-world scenarios. Our linear runtime in terms of number of gates is similar to many synthesis and analysis tools, since our tool and other tools require the parsing of every gate in the design.

Additionally, the algorithm is trivially parallelizable. Our initial implementation is sequential, but in the future it could be made parallel if necessary. We do not do any form of directed testing or targeting of specific rows in truth tables. We go with uniform randomness because any other method of randomness would be better for an attacker and worse for us as the security engineers.

One interesting lesson learned from our experiments was that false positives tended to be consistent (or even predictable) and did not occur due to the randomness of our sampling methods. We anticipate that the few false positives we do encounter will bear similarities to each other, perhaps allowing for easier recognition. Some examples of benign weakly-affecting wires (potential false positives) could be the most significant bit of a large counter or an input to an exception recognition circuit for a particularly unusual exception. These circuits are semantically similar to backdoors, because they react to a specific rare case. For example, consider the somewhat contrived case of a floating point divider that throws only a single exception, caused by a divide-by-zero error. Then for the data input representing the divisor, only the value zero invokes the exception-handling logic. Thus, the exception-handling logic is nearly unused, and that input is a weakly-affecting input. Many other such examples exist in moder microarchitectures, and to experienced designers, we believe they will be relatively obvious. For example, consider a performance counter that records an overflow after it has reached $2^{32} = 1$. This looks just like a ticking timebomb trigger and throws an easily recognizable false positive. Another similar example could be a large CAM (content addressable memory), wherein a cell is only activated when its search

value is provided. If this is, for instance, a 32-bit search input value, then each CAM cell contains a 32-bit comparator. When looking at the results of FANCI, there will be a group of weakly-affecting wires for each CAM cell, and they will all look identical. This type of pattern should be easily recognizable.

We expect that the existence of these benign circuits in designs should not pose much of a problem for security engineers, because counters, exceptions, CAMs, and so forth are easily recognizable in code review and even by their FANCI signatures. Simply being aware of this problem should be enough in most cases to render this problem as a non-issue. Nevertheless, as an attacker, one might be motivated to include many such benign but deceptive circuits to increase the false positive count. A larger false positive count could increase the time security engineers need to spend on analysis and could possibly distract them from more relevant circuits. The challenge from an attacker's point of view is that each of these false positives requires a costly circuit, and so building an entire design this way would likely be impractical. Additionally, these types of circuits in practice tend to have obvious architectural purposes, so adding hundreds or thousands of them would be a dead giveaway in code review. For example, including a large number of exception handlers that serve no apparent purpose would be a likely source of concern during code inspection.

Our hypothesis was that in real designs (*i.e.* designs that one might buy as IP), even malicious designers are forced to follow common design conventions and design reasonably efficient circuits. We believe that this is the reason we did not find a significant number of false positives in any of the designs we analyzed.

8.5 Evaluation of FANCI on an Out-of-Order Microprocessor Core

In order to study FANCI on a larger and backdoor-free design, we conducted a case study using the FabScalar microprocessor core generation tool [Choudhary *et al.*, 2011]. FabScalar is an HDL code generator that produces processor cores given a fixed set of parameters. The particular core we choose to use is a moderately-sized, out-of-order core with four execution units and does not contain backdoors.

The core we analyze has a total of 56 modules. The modules contain about 1900 distinct wires on average, with the largest module containing slightly over 39,000 distinct wires. This largest one is abnormally large for a single module containing primarily combinational logic. However, as this is an auto-generated design, it is understandable. While the overall design is larger than any of the modules from the TrustHub suite, and larger than typical third-party IP components, many of the individual modules are on average around the same size as modules in the TrustHub suite.

We were able to analyze 54 of the 56 modules in FabScalar using 2^{15} row pair samples per truth table. The two largest modules are outliers and would take significantly longer (we estimate 3-10 thousand hours on a single laptop core). These could easily be analyzed in a commercial setting on a compute cluster. Additionally, many software optimizations could be applied prior to commercialization.

As expected, we did not detect any false positives in the benign FabScalar core with the triviality and mean heuristics (we did not experiment with the other two heuristics due to lack of time). To garner some further intuition for how our heuristics look for wires in benign hardware, we construct a histogram of a typical FabScalar module (shown in Figure 8.8). There are two big spikes at 0.5 and 0.25. We see several other spikes. The reasons for spikes is because semantically similar wires tend to have similar values, as we saw in the example of a multiplexer. For this module, there are no suspicious outliers, with all of the values being at least 0.01.

8.6 Red Team/Blue Team Stress Testing of FANCI

While the controlled experiments to test the effectiveness of **FANCI** yielded excellent results, we consider it interesting to stress test the tool in a less controlled and more demanding environment. For this reason, we performed a red team/blue team experiment as part of NYU's 2013 Embedded Systems Challenge (ESC), where several teams from both the United States and around the world (specifically Europe and Asia) tried to defeat FANCI. The tool performed well, catching all of the stealthy attacks and even a few of the non-stealthy (frequently-on or always-on attacks) [esc, 2013; Waksman *et al.*, 2014]. While

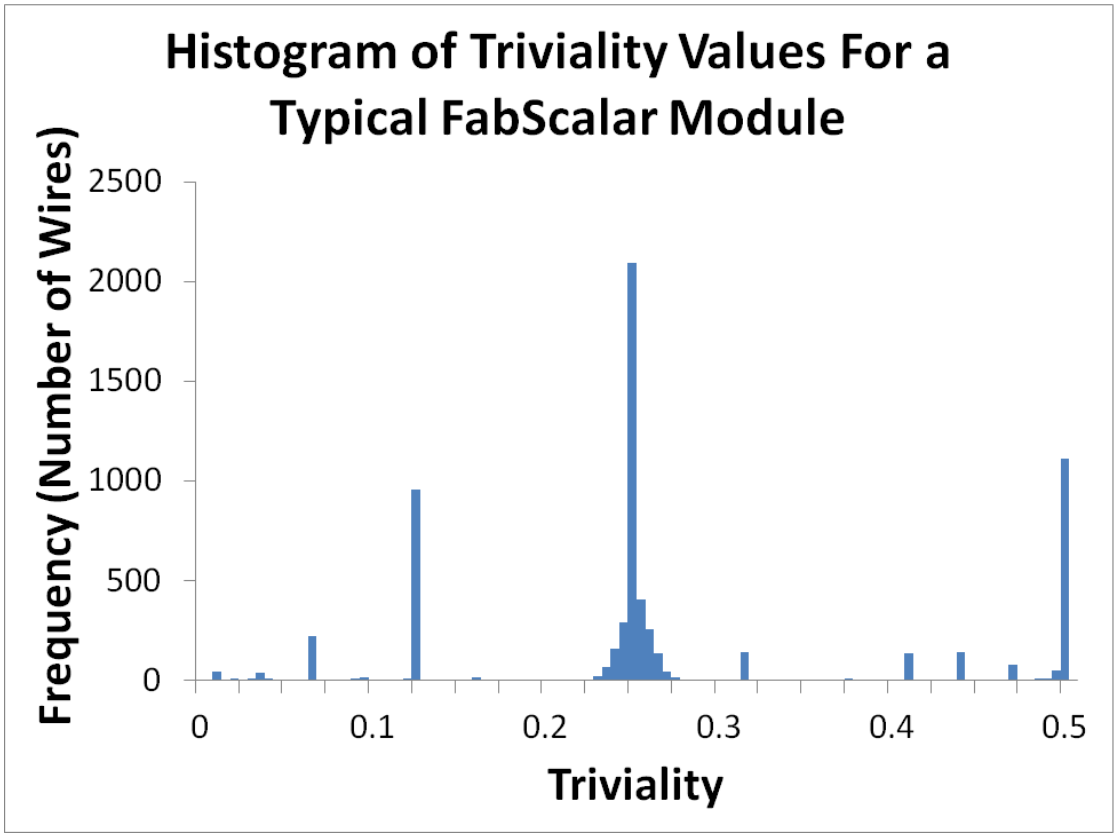

Figure 8.8: A histogram of the triviality values for wires in a typical FabScalar module, called the CtrlQueue. The two biggest spikes occur at around 0.5 and 0.25. There are no major outliers.

FANCI would not normally be expected to detect frequently on backdoors, sometimes even frequently-on backdoors use incidentally stealthy or abnormal logic for trigger recognition.

While the rules of the contest were not entirely formalized, teams were encouraged to attack any and all axioms of FANCI, and the blue team was put under extreme time constraints (only a few hours to catch backdoors) so as to press FANCI as much as possible. While several of FANCI's axioms were targeted by attacks, FANCI displayed resilience. All highly stealthy attacks were detected with relative ease. We found overall that the best way for attacker's to go after FANCI in this contest was to either use a frequently-on backdoor or to use a pathologically pipelined design. In the first case, frequently-on backdoors are not stealthy. Hence, FANCI will not flag them as such. In the second case, extreme pipelining

makes it harder to understand the data and stress-tests those portions of the tool prototype that are least developed (*i.e.,* parts that unwind memory loops). We include further details and discussion for the three most successful red teams from the competition.

• **Red Team A:** Red Team A created three independent backdoors (in three different but similar versions of a design), which highlighted different aspects of the FANCI tool and algorithm. This team created two frequent-action backdoors (which were not detected) and one stealthy backdoor (which was detected). The stealthy backdoor used a 66-bit cheat code that comes from a pseudo-random input source. This backdoor highlighted the importance of using intermediate wires in our algorithm, which was one of the main contributions of FANCI. If FANCI only computed the impact of module inputs on module outputs, this type of backdoor might not be caught, but since we analyze all intermediate wires, noticing a 66-input intermediate AND gate is easy. The two frequent-action backdoors were naturally not caught. One was turned on by the *reset* signal, and one was turned on frequently but only during the negative clock edge. Both of these were frequently enough on that they yielded no low control values and were not stealthy. These types backdoors highlight the need to use a minimal degree of validation testing in addition to FANCI. They also illuminate the fact that signals like reset and the clock, which might be given less attention during validation testing in practice, are just as relevant as any other signal when considering intentionally hidden backdoors.

• **Red Team B:** Red Team B submitted three backdoors as well. This team produced a variety of backdoors on independent baseline designs. The first backdoor was designed into a DES accelerator and was built off of an aggregation of small finite state machines (FSMs). They used the state machines to control round scheduling in an incorrect fashion and thus weaken the strength of encryption, which would expose the ciphertext to software-based attacks upon data release. This was a case where we were able to detect the trigger circuit because of the combinational logic that controlled the state machines, even though we ignore the actual state (flip flops).

The second backdoor from Red Team B was a similar round-scheduling attack, this time perpetrated against XTEA. This attack acquired its stealth by counting the number of one-bits in the input, which is a good way to generate high stealth, due to the non-uniform

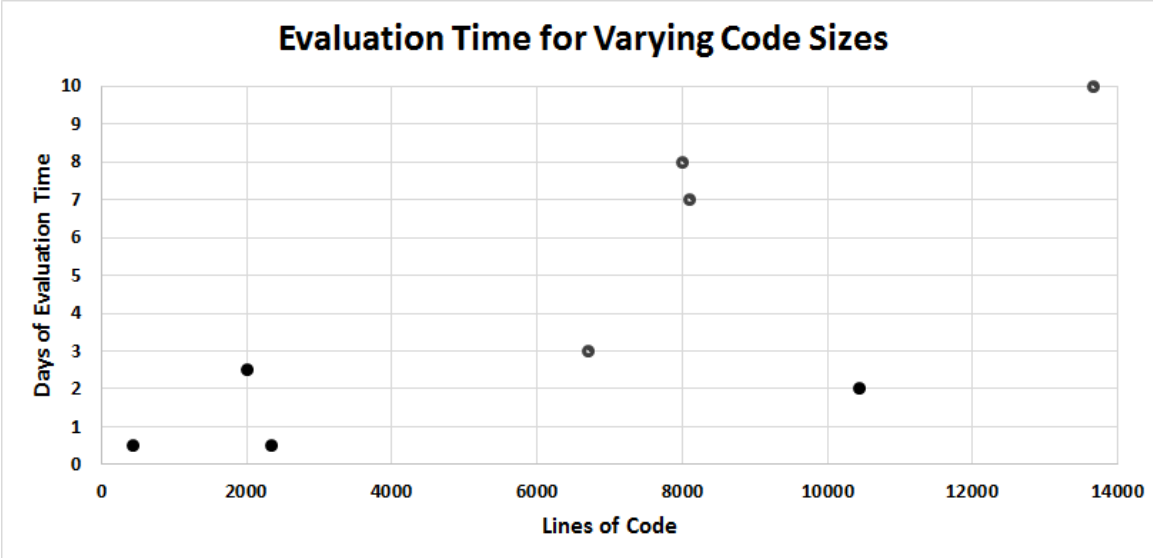

Figure 8.9: Trade-off between lines of code analyzed and runtime. The black points denote designs that finished completely in the contest time frame. The lighter points represent designs that were analyzed partially for the contest. For those points, the full runtime is estimated.

distribution of one bits across a bounded set of integers. However, the combinational circuit for doing the counting was flagged.

The third backdoor also attacked cryptographic hardware, this time a cellular automata PRNG. This one was a sequence cheat code where the malicious user needed to supply one bit of the trigger at a time. This attack would be normally difficult for validation testing to catch, because it hides on the negative edge of the clock (similar to one of the Red Team A attacks). However, since FANCI ignores clock timing, it can detect the trigger logic regardless of clock edge.

• **Red Team C:** Red Team C produced one backdoor, hidden in a stream cipher module. This attack was similar to one of the attacks suggested in the original FANCI paper [Waksman *et al.*, 2013b] and helped to demonstrate the need for basic validation and oversight when applying FANCI to untrusted designs. The attack is a combination of a frequent-action backdoor and a pathological pipeline backdoor. The trigger fires frequently but not all the time, meaning the stealth scores are not particularly low. The design is also heavily

pipelined, with roughly one flip flop between each pair of complex logic gates for critical paths in the design. Looking at the gatelist, it is immediately obvious that the design has been compromised due to the irrational amount of pipelining and poor latency. However, identifying the exact behavior of the backdoor payload in this setting can be difficult, and FANCI does not detect this payload readily or easily. Going after this type of attack in practice requires either rigorous validation tests or oversight from an integration engineer (to notice the pathological design style).

Overall, the contest served as an informal demonstration of the power and effectiveness of FANCI against sophisticated groups of attackers. We include a few observations and takeaways based on the results of the contest and our experiences.

• **Runtime and Scability** Figure 8.9 shows the runtime of the tool as a function of the number of lines of code in the various designs, using one primary design from each red team. Naturally, FANCI runs slower on larger designs, but the slowdown is more or less linear, which makes the analysis feasible. These tests were done on a single core of a commodity machine.

• **Attack Categorization:** A positive result of the contest was the discovery that many of the red teams designed attacks very similar to the types we anticipated when first designing FANCI. In [Waksman *et al.*, 2013b], we mentioned three general attack avenues against FANCI: frequently active (non-stealthy) backdoors, heavily pipelined backdoors, and false positive flooding. While the third option was not employed by the red teams, the first two were used by multiple teams. This evidence supports our belief that FANCI and validation testing should to be used together synergistically. Ideally, validation testing should be designed with the assumption that FANCI will detect anything stealthy. This would allow validation teams to focus their efforts on other avenues, such as some of the attacks we saw that target reset or the negative clock edge.

• **Algorithm vs. Implementation:** While the contest did not expose any deficiencies in the FANCI algorithm, the tool itself was stressed in some cases. Two issues stand out. First, runtime became an issue for large designs. Some modules would have taken more than the given three days to analyze, and so incomplete analysis runs were done. The tool is configurable for this, allowing for hasty passes. However, in the future, parallelization

could do a much better job of alleviating this problem. The second issue is the way the tool handles pipelining. The core of the tool works on combinational logic, so flip-flops have to be dealt with. We believe the best way to handle flip-flops is to treat them as identity gates, so that simply inserting dummy flip-flops does not hide stealthy logic. On the other hand, this creates loops in the logic, which have to be dealt with. For most cases, our tool currently treats flip-flops as a barrier and does not analyze past them. This did not prevent us from catching any stealthy backdoors in this contest, but it made manual analysis more difficult. Improving the tool for this case in the future would be beneficial and likely necessary for commercialization.

8.7 Security Discussion and Guarantees of FANCI

We briefly discuss a few of the key security properties regarding **FANCI**.

• **FANCI** relies on the fact that backdoors use weakly-affecting wires. This is because backdoors need to be stealthy, *i.e.* well-hidden from validation testing. The more well-hidden a backdoor is, the more likely it is to be caught by FANCI, because well-hidden backdoors have lower control values. On the other hand, the less well-hidden a backdoor is, the more likely it is to evade FANCI. Fortunately, less well-hidden backdoors are more easily caught by standard validation practices. For example, a poorly-hidden backdoor that turns on frequently (and thus has high control values) will go unnoticed by **FANCI** but would be caught during basic validation testing. It is provable (see Section 8.8) that for a fixed-depth combinational circuit path, achieving a given level of stealth requires a correspondingly low control value for one or more of the inputs.

• **FANCI** does not entirely remove the need for commonplace code inspection/review practices. As a motivating example, consider an attack in which a malicious designer includes thousands of backdoor-like circuits, *i.e.* circuits that have triggers. By giving only one or a few of them useful payloads, the circuit would appear to **FANCI** as one with many backdoors, producing a large number of what one might consider false positives and making analysis take an excessively long time. Of course, in addition to the area bloat this would cause, it would be clear in basic code inspection that this was a contrived design. Nonetheless, we consider this to be a relevant case that anyone using **FANCI** should be aware of. **FANCI** specifically targets small, well-hidden backdoors, which are the type that are able to evade testing and code inspection. It is not well-suited for catching backdoors that are large and relatively obvious.

• Given knowledge of how **FANCI** works, it would be possible for an attacker to attempt to deceive the tool's analysis. Since an attacker cannot create a backdoor without weakly-affecting wires, an attacker might instead try to hide a backdoor amongst other weakly-affecting wires, hoping to cause false positives. However, in practice these types of circuits (such as comparators and exception handlers) tend to have architecturally obvious functionality. If the backdoor is the only flagged wire that does not have documented functionality,

it stands out in code inspection. It remains theoretically possible to imagine such an attack, though it might not be realistic. Consider for example a divide-by-zero exception handler in a floating point unit. Consider that an attacker puts a backdoor in the unit that fires when an instruction says to divide by seven. The divide-by-zero exception is easily recognizable as conventional, while it would be suspicious to special-case on divide by seven, especially if its functionality is not documented or known to the architects. **FANCI** would likely flag both of them, but simple code inspection would immediately recognize the true culprit.

A qualitative but important property of our approach is that it behaves well with respect to common, reusable structures. In common designs, especially large designs, much of the circuitry is spent on standard, reusable components, such as CAMs, RAMs, FIFOs, decoders, encoders, adders, comparators, registers, etc. We do not have issues with false positives for these common types of structures, and a big part of that has to do with our heuristics. When choosing false negatives, we look for outliers, as mentioned previously. In these standard structures, there tend to be no outliers due to symmetry. Consider a CAM with 32-bit data entries. For each entry, there is a 32-bit data comparator which includes some very low control value dependencies (on the order of $\frac{1}{2^{32}}$. However, due to symmetry, each of the comparators is identical (or nearly identical), thus leaving no outliers to serve as false positives.

Our approach also works very well against stateful (*viz.* sequential) backdoors, as we saw in some of the TrustHub benchmarks. By simply treating latches as identity gates, we can ignore state and still uncover low control value dependencies. Hypothetically, a sequential backdoor that makes use of an extremely large and deep (contrived) state machine might be able to avoid outliers. However, as we saw in Figure 8.5, practical backdoors have relatively small lengths.

8.8 The Mathematics of Circuit Stealth

The underlying intuition behind the FANCI approach is that stealth is a necessity when creating a design-side backdoor in the presence of validation tests. This intuition and our empirical evidence have thus far supported the idea have there are strong motivations to use stealthy circuits and that these stealthy circuits need to make use of wires with low control values. We briefly express the relationship between control values and stealth which corroborates this intuition and evidence.

Theorem 2. The Stealth/Control Duality Theorem. *In a fixed depth, combinational, digital, boolean circuit, high stealth (above* $1 - \frac{1}{\epsilon}$ *for a given* $\epsilon > 0$*) requires at least one low control value (below* δ *for a corresponding* $\delta > 0$*).*

Intuition: The intuition for this theorem is relatively clear. Stealth and control are loosely opposites of each other. High stealth means that a behavior rarely happens, which means the control value of inputs on the behavior must be low. As stealth increases, control correspondingly decreases.

Relevance: This theorem explains the empirical results using the **FANCI** tool. Since attackers are forced to use high stealth, they are forced to use low control values and thus make the application of **FANCI** easier.

Proof: Consider a fixed depth circuit with all the above properties, depth $d > 0$ and $g > 0$ gates that form a combinational circuit (*i.e.* a connected, directed acyclic graph). Consider that one edge in this graph is a trigger wire T which must take on the value of 1 in less than an ϵ fraction of all possible cases so as to achieve a stealth level greater than $1 - \frac{1}{\epsilon}$. Consider the restricted boolean function with the input set consisting of the subset of inputs that contain T in their fan-out tree and with $\{T\}$ as the output set. The corresponding graph represents a subgraph of the original graph, and the stealth of T is unchanged. For each element e of the input set (each node in the graph that has an in-degree of zero), we can denote the two Shannon cofactors as $\mathcal{F}_e$ and $\mathcal{F}'_e$. The boolean difference between $\mathcal{F}_e$ and $\mathcal{F}'_e$ is at most 2ϵ, with the extreme case being the case where $\mathcal{F}_e$ is the always-zero function. Therefore, if there exists a wire T with stealth greater than $1 - \frac{1}{\epsilon}$, there exists at least one corresponding wire e such that the control value of e on T is at most 2ϵ. This

satisfies the theorem by setting $\delta = 2\epsilon$. ■

We note that not all circuits are combinational. In fact, pipelining is the most clear way to break up a backdoor and lessen the use of extremely low control value wires. There are several factors that have made pipelining insufficient to effectively bypass FANCI, and we briefly summarize those factors.

The first issue is that pipelining has little impact if it does not result in loops. In other words, if a flip-flop bank only cuts a combinational circuit path in half, that has no real impact on control values. We can simply pretend the flip flops are not there (because timing does not matter for our purposes) and run the same analysis. This has worked well for us empirically.

A second issue is that even if the pipelining creates loops, if the loops are not highly complex, then the analysis is not forced to change significantly. For example, if the loop can be broken by severing one connection, then we can simply perform two analysis jobs, one on each side of the severed connection. This makes the analysis task slightly more difficult but not radically different.

A final issue is that in the case of extremely complex, pathological pipelining, the design begins to look contrived. As we have seen with some example circuits, FANCI can be made to fail by using extremely contrived pipelining methods, but the design looks unrealistic to any experienced engineer looking at the source code. For an attacker to come up with a design that is sufficiently pipelined to avoid FANCI while also being stealthy enough to evade all likely validation tests and still appear ordinary enough to pass basic code inspection is a tall order that we believe raises the bar to a high level.

Chapter 9

Disabling Backdoor Triggers Dynamically at Runtime

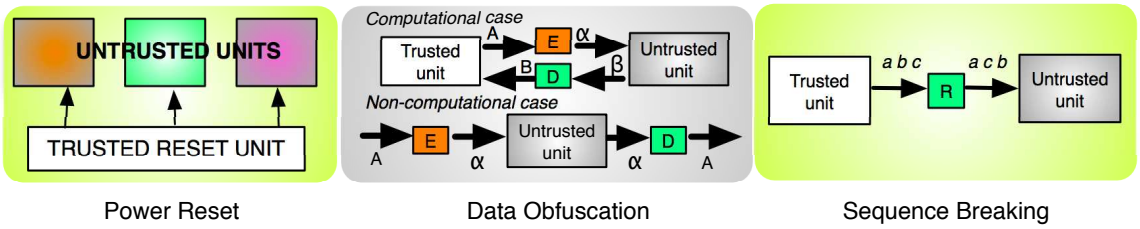

Figure 9.1: An overview of our three methods for trigger obfuscation.

Even if hardware designs cannot be trusted, it is possible to disable hardware backdoors by taking action at runtime. Toward this end, we proposed *trigger obfuscation*, a novel approach that allows the triggers of backdoors to be disabled before any payload has been achieved. This means that if we do not trust the design, either because functional analysis is imperfect, because we do not trust our own insider designers, or for any other reason, we can fall back on trigger obfuscation as a runtime protection system.

Our approach is to include security circuits within the design itself that enforce security at runtime. We add these circuits after the design has been completed and validated, so that even a malicious insider on the design or validation teams cannot compromise them. This is an important application of the Principle of Last Action.

Our key insight is the following. A stealthy hardware backdoor always contains a trigger, which is the unique signal that turns the module from benign mode to malicious mode and enables the backdoor payload. Our idea is therefore to scramble signals at the input interfaces of hardware modules to prevent backdoor triggers from going off. Since payloads are dependent on triggers, if the triggers do not arrive, the malicious payloads cannot be delivered. We discuss how the three types of digital triggers discussed previously (ticking timebomb, single-shot and sequence cheat codes) can be scrambled using this approach.

- *Power resets* protect untrusted units against ticking timebombs. A ticking timebomb, as mentioned before, goes off after a passage of a certain amount of time, say after 2^{40} clock cycles of operation. This can be easily implemented by an attacker with a 40-bit down counter. The timebomb will not go off during design validation because validation is carried out for much shorter time scales due to the slowness of validation testing and time-to-market constraints. Typically, validation suites run in the KHz range compared to real

hardware that runs in the GHz range. As such, validation tests are run for a small number of cycles, say 10^7 cycles per test. Given these parameters and constraints, we can see why a malicious engineer on the design team who has insider information on the duration of the validation tests can easily use this information to make the backdoor trigger fire well after the validation period.

Our solution to mute this trigger is to prevent circuits from knowing that a certain amount of time has passed since start-up. We ensure this by frequently powering off and on (or resetting) each unit, causing all microarchitectural data to be lost. The frequency of power resets is determined by the validation epoch. We know and trust the circuit under test to be free of timing backdoors during the validation epoch (otherwise it would not pass any validation); so resetting power before the end of the validation epoch ensures that all state in the unit is wiped clean and the unit on longer has "memory" of previous computations. For instance, if the attacker is using a counter to trigger the backdoor, that counter will be reset to its initial value and never get a chance to hit the trigger.

Now the question is: if we are constantly resetting power to the unit, how can we make forward progress at all? Many common hardware modules behave transactionally, meaning that the module applies a function only to its inputs and does not need to maintain long-term state. However, some hardware requires continuity of state across reset epochs. To ensure forward progress in this case we simply identify and store all architectural state necessary for forward progress outside the unit before the reset. From an implementation perspective the reset idea can be implemented using "power gating," circuits, a technique to limit power consumption where units are reset during periods of inactivity.

With this approach, how can we be assured that the attacker does not exfiltrate the time counter (or some proxy that encodes the passage of time) through the state necessary for forward progress? This is not possible due to our axiom that the unit does not produce incorrect results for the duration of the validation epoch. If time information is illegally exfiltrated during validation, just one good validation engineer on the entire design team is sufficient to notice and detect this violation. A final move by the attacker could be to store the trigger in non-volatile state. This is a clever move but somewhat impractical because the design process for creating non-volatile hardware is so different from regular

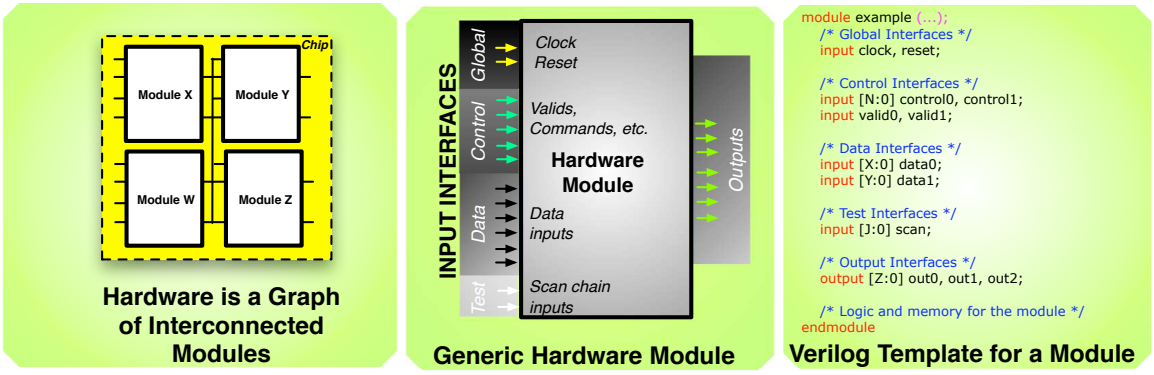

Figure 9.2: An overview of how a typical hardware module looks in terms of its input and output interfaces.

hardware that this move would trigger off many different alarms at various stages in the design process. As such, this may not pose a significant risk to most hardware units except those that already require non-volatile memory by design (such as TPMs). Solutions to this problem are discussed in the original paper [Waksman and Sethumadhavan, 2011].

- The second category of backdoor trigger is the *single-shot cheat code*. This type of trigger is delivered as a data value thorough one of the hardware input interfaces. Our solution here is to scramble the inputs supplied to a hardware module. Scrambling the inputs silences backdoors because the attacker now cannot anticipate the trigger. The challenge here, however, is to perform computation in the unit without unscrambling the input value.

  Our solution draws on domain-specific applications of homomorphic encryption (or homomorphic obfuscation, since as we will discuss shortly there is no need for cryptographically strong encryption in hardware settings). We provide a brief introduction to homomorphic functions and then explain our solution with an example.

  A homomorphic operation is one that preserves the structure of another operation. An operation f is considered to be *homomorphic* with respect to another operation g if $f(g(x), g(y)) = g(f(x, y))$. One simple example of this is when f is multiplication and g is the squaring function. For instance,

$$x^2y^2 = (xy)^2$$

If the functionality required of a hardware module is to compute the square of a value, we can obfuscate the input x to that unit by multiplying it by a pseudo-random value y. The squaring unit then computes the value $(xy)^2$. Then to decrypt, we only have to divide by the constant y^2 to get back x^2. Since we permuted the input space in a homomorphic (structure preserving) way, we did not undermine the usefulness of the squaring module. More generally, if our obfuscation function is homomorphic over the computational function, then the computation can be done on the data while it is encrypted, which is exactly what we want. This is one specific application of homomorphic functions, which have been used more broadly in software settings and in theory literature [Gentry, 2010]. If we can create low overhead homomorphic equivalents for typical functional units, we can improve protection against hardware backdoors.

It is also worth noting that for many hardware units, the work performed in the hardware module is an identity operation (moving data or storing without changing it, *e.g.,* routers, caches), in which case any operation can be used for obfuscation. In those cases, we simply use a bitwise XOR operation with a changing random number (an one-time pad) because it adds only one gate to the path delay.

- The third and final type of backdoor trigger is the *sequence cheat code*, which we protect against with a method called sequence breaking. Here the trigger is received as several pieces of data over multiple cycles. Our solution is to benignly reorder sequences of inputs. If the attacker cannot pre-determine the order in which transactions will be supplied to the hardware module, the attacker cannot trigger a backdoor with any significant probability. The main constraint here is to ensure that changing the order of transactions does not change the functionality of the hardware unit.

For a realistic example, consider a memory controller with a backdoor that is triggered by a sequence of fifty loads and stores that must come in a pre-chosen order. By changing the order of those fifty loads, we make it unlikely that the correct trigger will ever be supplied. The intuition for why it is easy to make the probability of attack failure effectively 100% is that the trigger ends up being one pseudo-random choice out of the set of all possible permutations of memory instructions, which is a factorially large set.

It may not be possible to reorder input transactions to a unit in all cases. For example, a

malicious user may be able to contrive a sequence of inputs to a memory controller that are not amenable to reordering without violating correctness requirements such as consistency or coherence requirements. For such cases we propose adding dummy events. Inserting random dummy events such as non-binding loads to the memory controller is sufficient to break any pre-determined order for a trigger.

9.1 Application to a Modern Microcontroller

As a practical case study and to learn from the engineering and implementation of our methods into real hardware, we performed an in-depth case study with a microcontroller to better understand the difficulties and limitations associated with applying trigger obfuscation to modern hardware. Our goals in this study were three-fold: 1) to design an algorithm for assisting security engineers in their tasks, 2) to better understand any corner cases that might arise during the security engineering process, 3) to determine how well different security approaches work together, and 4) to quantify the overheads associated with applying security to real hardware.

9.1.1 Motivation for Building the TμC1 Microcontroller

We first outline the existing defensive techniques that protect against design-level hardware backdoors, which include both our own methods and previously existing methods. These methods can be applied module-by-module *á la carte*. Other solutions may be proposed in the future, but these are the prominent methods existing today and are sufficient for us to protect against design-level backdoors.

These methods include the three components of trigger obfuscation (rapid resets, sequence reordering and data encryption/obfuscation), as well as pre-existing methods: formal verification, exhaustive validation, homomorphic encryption and dual-modulo redundancy. We discuss each method, including the pros and cons, as well as the necessary assumption and domain of applicability.

• **Formal Verification:** The goal of formal verification is to prove correctness of a design during design-time, assuming trust in a formal specification. Formal verification can be applied to small systems or parts of systems; the cost scales exponentially with the complexity of the system. The cost of writing the formal specification itself tends to increase design costs and can be fundamentally hard depending on the situation.

• **Fully Homomorphic Encryption (FHE):** FHE [Gentry, 2010] is a technique for encrypting both the data and the operations within a program. The technique can be mapped to hardware by thinking of circuits as operations. Even a malicious insider cannot

Table 9.1: A summary of seven prominent methods for protecting against hardware backdoors. This table outlines coverage, performance overheads and the inherent trust models at a high level.

Type of Defense	Engagement Stage	Covers All Trojans	Performance Overhead	Required Trust Assumptions
Formal Verification	Design Time	Yes	None	Moderate
Exhaustive Validation	Design Time	No	None	Low
Homomorphic Encryption	Runtime	Yes	Very High	Low
Dual Modular Redundancy	Runtime	Yes	Low	Lowest
Rapid Resets	Runtime	No	Negligible	Low
Sequence Reordering	Runtime	No	Low	Low
Data Encryption	Runtime	No	None-Low	Low

Table 9.2: A Summary of seven prominent defenses against hardware backdoors. This table provides a further breakdown of the types of overheads incurred by each method.

Type of Defense	Performance Overhead	Area Overhead	Power Overhead	Human Overhead
Formal Verification	None	None	None	Very High
Exhaustive Validation	None	None	None	High
Homomorphic Encryption	Very High	Very High	Very High	Moderate
Dual Modular Redundancy	Low	High	High	High
Rapid Resets	Negligible	Negligible	Negligible	Negligible
Sequence Reordering	Low	Low	Low	Negligible
Data Encryption	None-Low	Low-High	Low-High	Low

compromise such a system due to the circuitry itself being encrypted. Any backdoor hidden in a circuit would have virtually no chance of being triggered, and if it were triggered its payload would be garbled. FHE is both a strong and expensive solution. We evaluate the possibility of applying FHE to protect against hardware backdoors. We find that FHE is

Table 9.3: A summary of the different trust assumptions made by prominent hardware backdoor defense methods. The table covers the range of common trust assumptions, such as a formal specification or the existence of trusted tools for physical synthesis.

Method	Spec.	Designers	Validators	Tools	Foundry	Testing
Formal Verification	Yes	No	No	Yes	Yes	Yes
Exhaustive Validation	No	No	Yes	Yes	Yes	Yes
Homomorphic Encryption	No	No	Yes	Yes	Yes	Yes
Dual Modular Redundancy	No	No	No	Yes	Yes	Yes
Trigger Obfuscation	No	No	Yes	Yes	Yes	Yes

orders of magnitude too expensive for practical use, but we put into perspective those costs and what it would mean to have FHE available in hardware.

• **Dual Modular Redundancy (DMR):** DMR is a classic technique for building reliable systems out of unreliable components [von Neumann, 1956], the idea being that the probability of two independent entities making the same error at the same time is very low. To apply DMR to backdoor protection, we acquire two different implementations of the same module with identical interfaces from two different supply chains, meaning there is roughly a 2X area and power overhead, as well as added design-time costs. DMR provides only error detection. Adding a third copy would allow for error correction, but detection is generally the target in hardware security.

• **Validation:** Validation testing (and/or verification) confirms that a design produces correct results for some limited test cases or period of time.

• **Rapid Resets:** This method, discussed in Section 9, resets power frequently to erase transient state and prevent timebomb triggers.

• **Sequence Reordering:** This method, discussed in Section 9, reorders microarchitectural events to prevent sequential cheat code triggers.

• **Data Encryption**: This method, discussed in Section 9, uses domain-specific homomorphic obfuscation to prevent single-shot cheat code triggers.

We summarize qualitatively the costs of the seven types of defenses in Table 9.1. We consider the three primary design metrics — performance, area, and power — as well as the human design/engineering effort, which can be a major bottleneck, especially when accounting for time-to-market constraints. Unsurprisingly, none of the seven techniques are without overheads or limitations. We also summarize the trust models in Table 9.3, aggregating rapid resets, sequence reordering and data encryption under *trigger obfuscation*.

9.1.2 ART: An Algorithm For Making Trigger Obfuscation Decisions

In this section, we present **ART**, a simple algorithm for helping security engineers to make decisions related to trigger obfuscation. The algorithm is essentially a decision graph, which determines when to use trigger obfuscation (as opposed to older techniques) and in what manner it should be used.

The basic idea is that we need to assign protections to each module in an untrusted design. **ART** is a simple decision algorithm that does this. It chooses the best combination of techniques from the seven known techniques listed in Table 9.1. Concretely, there are $2^7 = 128$ different choices for the defense of any given module, because for each of the seven methods in consideration, we can either use it or not use it. Overheads such as area and power for each defense can vary depending on the module. Further, subjective opinions on the necessity for security coverage may also vary: a given organization may only care to cover against a subset of possible attacks. Security architects have to decide on their needs and constraints before they can choose the right solution.

The **ART** algorithm is a straight-forward and intuitive way to choose from the space of 128 choices for the case where complete coverage is desired. If only partial coverage is needed, some modules can be left out. Constraints on any of the dimensions of overhead (such as timing and power) can vary within the qualitative bounds described in Table 9.1, with specific values being user-provided. We make conservative assumptions about the relative costs of the methods. Given these relative cost relationships and the necessity for complete coverage, we notice that only five of the 128 choices can be the most economic choice for a given module. We know this for the following reasons:

Formal verification, FHE and DMR each provide complete coverage when applicable and

thus need not be mixed with other techniques for a given module. Using excess protections can only increase overheads. That yields three choices and leaves $2^4 = 16$ options remaining for the other four techniques. Since validation does not provide complete coverage, it must be combined with dynamic techniques. If validation can be applied to all interfaces of a stateless module, then it only needs to be combined with rapid resets (choice four in Figure 9.3). Otherwise, all three dynamic techniques are required (choice five).

That exhausts our possible choices, so we know that our algorithm should always pick one of these five choices. Which of the five is best depends on the module and the constraints of the design setting.

The ART algorithm is the formalization of the natural line of questions a good engineer should ask. For each module and each option, we ask whether or not it meets our constraints. For instance, we might or might not have a formally verified module at hand. We might or might not have a method for performing FHE. For each module, we ask the question: *Which techniques have practical implementations for this module?* We ask these questions to the user in an order that allows us to reach all five valid choices deterministically, and we proceed based on if the answer is *yes* or *no*.

Figure 9.3 shows the decision graph. The user provides the yes/no answers for each individual module, and the decision graph informs the user which defensive mechanisms to apply. The user's inputs are based on the module in question and the constraints of the design setting, such as power and area budgets or the availability of a formal specification.

For example, if a formally verified module is available, it will be used, and no further protections are necessary. On the other hand, consider an on-chip router node with 1-hot encoding as a third-party IP module. A possible user scenario could be the following. A formal specification is not available, FHE and DMR violate the power budget, the 1-hot encoding and valid bits can be exhaustively validated, and the data packets can be encrypted with XOR encryption within the power budget. The module is not stateless, and packets arrive in order, but they do not have sequential dependencies. Then the algorithm will go through states $1 \to 2 \to 3 \to 4 \to 5 \to 7 \to 8 \to 9 \to 10$ and complete at step 10.

This is logic that an astute engineer might go through naturally, but ART formalizes the order of priorities so that it can be applied universally.

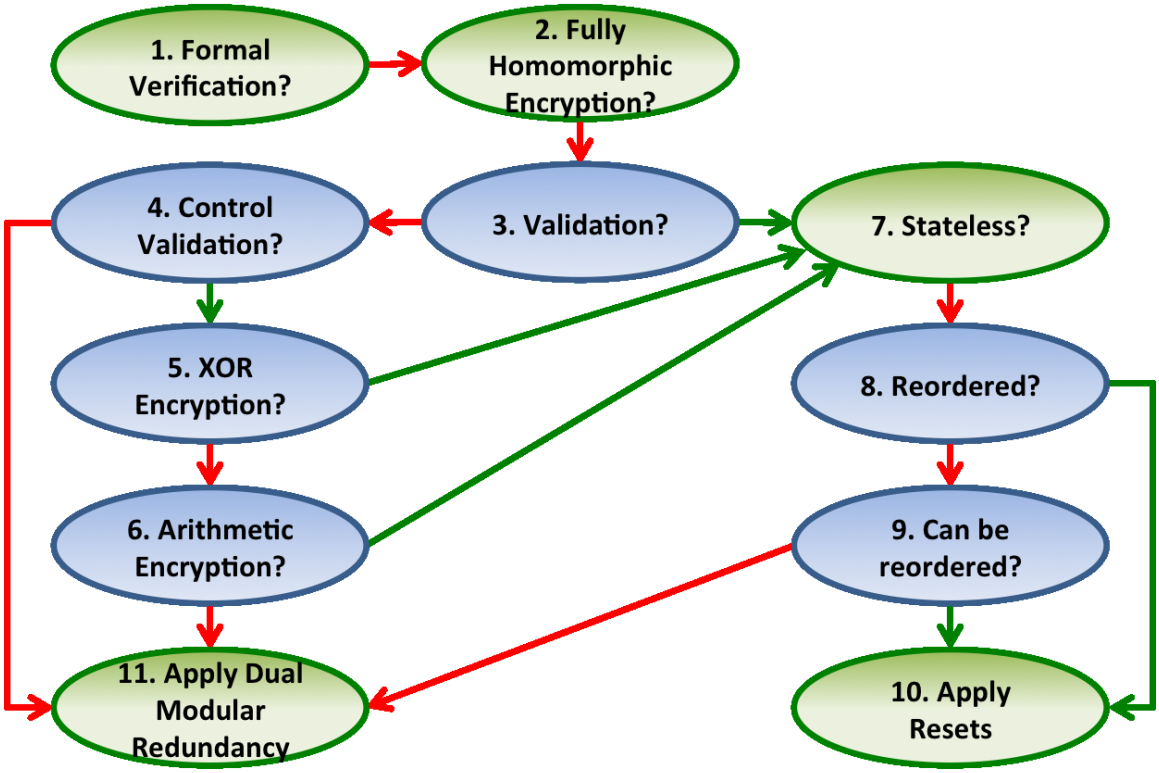

Figure 9.3: The steps of the **ART** algorithm. Green arrows signify a "yes" answer. Red arrows signify a "no" answer.

It is guaranteed that if one of the five valid choices meets the user-provided constraints then such an answer will be provided. For each choice, all backdoor types are covered. If the user does not have a fixed budget and simply wishes to minimize certain constraints, such as power, then traditional design exploration strategies (such as binary search) can be applied on top of ART.

9.1.3 Caveats and Limitations

We consider a few caveats and limitations of the **ART** algorithm for decision-making.

(1) ART is not the only possible algorithm. Our decision graph could be arranged in a few different ways to get the same results. However, it is the result we care about, and all such algorithms would yield the same results, so they are equivalent. We claim only that the techniques we have incorporated into our designs are sufficient to achieve coverage

against digital design-level backdoors and that the ART algorithm always yields a solution that fits within user-defined constraints if such a solution exists.

(2) The ART algorithm does not generate code. It is a formalization of the decision-making process an engineer has to go through. Given the results of ART, an engineer has to apply the chosen techniques to the design at hand.

(3) For all of the techniques we consider, there exists a common assumption of last action from prior work, which requires that security mechanisms/procedures are inserted/applied after the design phase has completed. From an operational security standpoint, it is crucial that this assumption is enforced. For example, the attacker cannot have the option to add a new module after the design has been finalized and the security mechanisms have been applied.

To evaluate the overheads of our technique we prototyped a microcontroller for a commercial AVR ISA [Waksman *et al.*, 2013a] that included all our security features. We found that by applying domain-specific solutions to each component in the design, we were able to protect each module without dramatic overheads.

9.1.4 Evaluation of ART Applied to the TμC1 Microcontroller

When we began developing TμC1, our intuition was that corner-cases and difficulties could arise in applying a variety of disparate and theoretical methods to real hardware. The most surprising result of our work is that the various protection schemes work synergistically with each other and can serve to patch over caveats in their original presentations. The whole turns out to be greater than the sum of the parts.

Based on manual analysis of a variety of designs, including memory controllers, computational cores and I/O controllers, we have determined that our methods can be applied directly to designs that are commonly used as third-party components. These types of designs tend to use standardized interfaces for ease of use. For example, memory controllers adhere to the DDR protocol. This makes it easier for us to apply protections to the interfaces.

As a case study and to demonstrate our methodology in full detail, we design and synthesize a new microcontroller called TμC1. This microcontroller implements the AVR

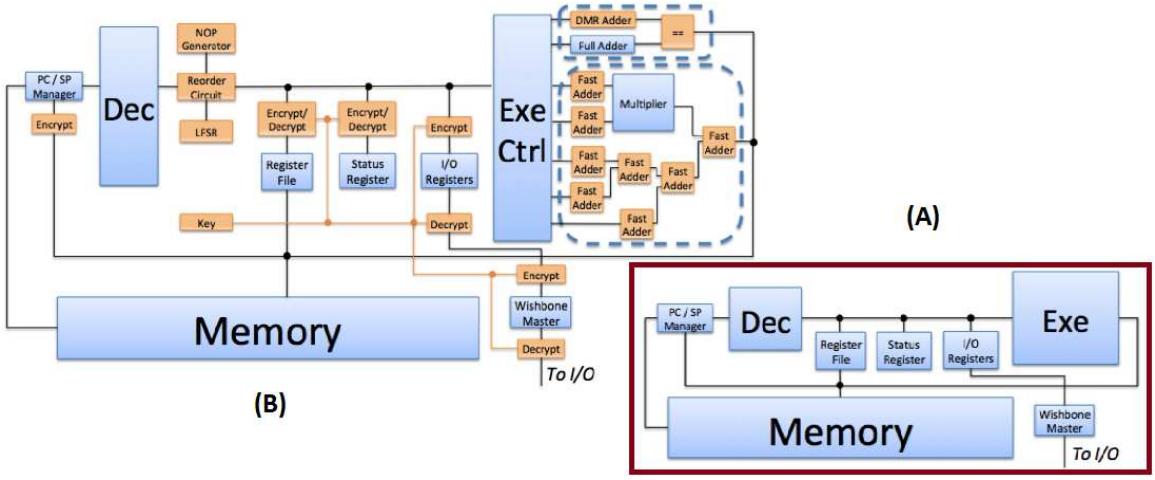

Figure 9.4: (A) The baseline microcontroller microarchitecture we use for our security design methodology. It is a simple processor, with on-chip memory and a wishbone master for off-chip I/O. (B) Layout of the TμC1 microcontroller (not to scale). Modules in orange (darker) perform security functions. Modules in blue (lighter) are part of the baseline specification.

ISA, a RISC ISA used commercially by Atmel. We choose to build a small and standard microcontroller because it is a hard and common use case for backdoor protection. In larger designs, our security structures would have comparatively lower overheads.

Figure 9.4(A) depicts the microarchitectural layout of TμC1. It is a simple, inorder processor, with a decoder, execution path and unified on-chip memory. There are four register files — data registers, I/O registers, status registers, and program counter/stack pointer holders. We also include a wishbone master to allow for bus communication. The choice of wishbone was fairly arbitrary. Other I/O protocols, such as USB, PCI, SATA, etc. could be made backdoor-free with an equivalent approach.

We next discuss the cost and scalability of different methods, beginning with formal verification. Formal verification involves aspects that are exponentially hard, resulting in poor scalability. However, formal verification is a lively field of study and more breakthroughs are always possible. For the rest of the methods, we can compute how they scale.

For Fully Homomorphic Encryption (FHE), our experiments show that FHE is not a practical option for security. In FHE, input data has to be encrypted before entering a unit and then periodically re-encrypted depending on the logic depth. Both of these operations

are too expensive.

The complexity of the encryption, which relies on large prime numbers, has a quadratic cost in terms of the bit width of the prime numbers. For a prime number size p and a logical depth d, the area and power costs of a single FHE gate scale as roughly $c(2^d p)^2$ where c is the baseline cost.

For a conservative estimate, consider the relatively small prime number size of 64 bits and the minimum possible depths of one and two. Table 9.4 shows the results of synthesis in 90nm technology. A modern server die is usually 300 mm^2, meaning that we could, for example, fit at most about a hundred AND gates with depth two. These results show that FHE is orders of magnitude away from where it would need to be for practical applications.

Table 9.4: Baseline Costs of FHE Gates

Gate Type	Usable Depth	Latency	Area	Power
AND	1	150 ns	.866 mm^2	3.08 mW
OR	1	176 ns	.856 mm^2	3.01 mW
AND	2	388 ns	2.97 mm^2	11.4 mW
OR	2	387 ns	3.04 mm^2	11.6 mW

The implementation of data encryption depends on whether or not the circuit being encrypted does work with combinational logic. Data encryption in the simple case – such as memories, I/O, etc. – requires only a single XOR gate and scales at constant cost. On the other hand, data encryption for combinational logic does not have constant cost. We consider the units we needed for our microcontroller.

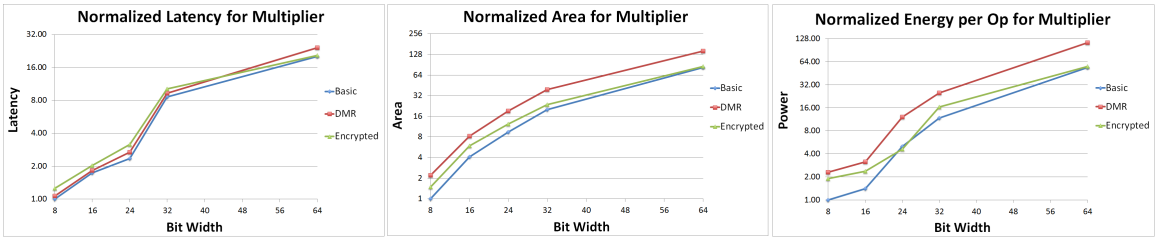

Figure 9.5: A scalability comparison of DMR and data encryption for multipliers. We consider latency (a), area (b), and energy per operation (c).

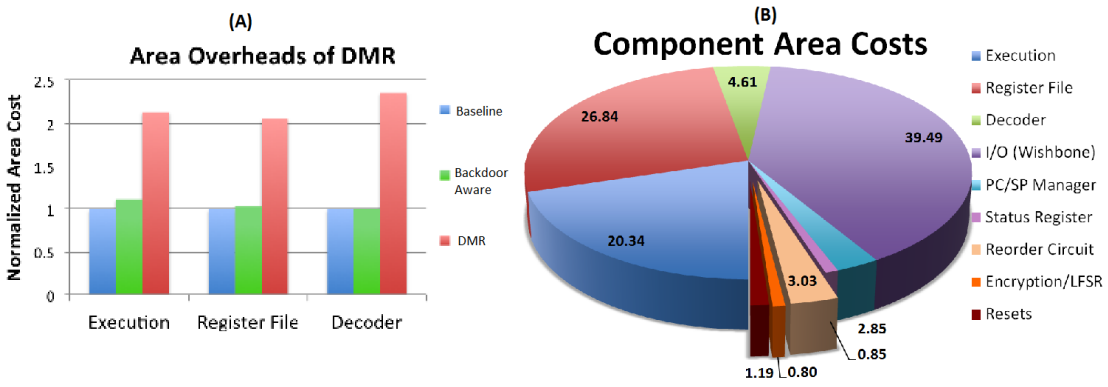

Figure 9.6: (A) Area impact of DMR on different modules, as compared against baseline costs and against optimal backdoor-aware choices. These measurements are taken in the context of the TμC1 microcontroller. (B) Breakdown by component of area costs for the TμC1 microcontroller as percentages of the whole.

The most substantial arithmetic module we need is a multiplier. Our encrypted version of the multiplier, which uses a polynomial computation for scrambling, requires seven adders in addition to the original multiplier. The adders are used to change the input values before the multiply and then alter the product after the multiply, leveraging basic properties of polynomials.[1] The area and power costs should scale roughly as $7n + cn^2$ because the adder logic grows linearly and the multiplier logic grows quadratically. For comparison, DMR requires two multipliers and a comparator, so the costs scale as $2cn^2 + n$. The expectation should be that the encrypted multiplier scales better, and this is supported by our results. Figure 9.5 compares the latency, area, and energy per operation scaling from $n = 8$ to $n = 64$. At the small end (small enough for exhaustive validation), DMR and data encryption are comparable and are about double the cost of a baseline multiplier. At the larger end ($n = 32$ to $n = 64$), DMR remains double the cost, but data encryption becomes the same cost as the baseline multiplier because the quadratic term dominates the linear term.

Applying Dual-Modulo Redundancy (DMR) requires two versions of a module and a

[1]The equations can be derived from the fact that multiplication distributes over addition. For brevity, these are left out.

comparator for each interface. The cost is determined by how wide the module is (how many interfaces) compared with how deep it is (how much circuitry is used for internal computation). If we consider a circuit with original area a and width w, the area and power of a DMR version scale as $2a + wc$ where c is the cost of a comparator. The latency scales as $a + c$ because the comparator is the only addition to the critical path. Figure 9.12(A) shows the costs of DMR in practice for some example modules. We see that the area overhead ranges from about 2.1 to 2.4. These measurements are taken in the context of the TμC1 micrcontroller and are compared against the overhead of using optimal backdoor-aware techniques, as determined by **ART**. These results show that DMR can be expensive and should be used judiciously.

Exhaustive validation can either be validation on the set of all possible control inputs (2^c cases for c control bits) or the set of all inputs (2^n cases for n total bits), depending on which exit point we use in the **ART** algorithm. The total cost of validation is roughly $v2^b + h$, where v is the baseline runtime of a single test, b is the number of bits (either c or n) being exhaustively tested, and h is the one-time human overhead of building the testbench. In our experiments, v is around 0.1 milliseconds on a desktop computer, so the largest validation run, which exhausts over the 24 bit execution control interface, takes about 28 minutes.

We synthesize our designs, including the microcontroller and all included security methods, in 90nm technology with ASIC libraries and measure area, frequency and power. Our core design synthesizes to 0.070 mm^2 at 91 MHz and 0.45 mW of dynamic power. Our synthesis measurements include all core components as well as memory and I/O control but not the cost of on-chip SRAM.[2]

We consider three different implementations of $T\mu C1$. The first is the fully secured TμC1, which is scalable to arbitrary data-path widths because of the use of scalable data encryption and DMR techniques. The second is a secure design that we know will not scale because it relies on validation of some interfaces that would grow too large. As we showed in Figure 9.5, data encryption techniques are not necessary for an 8-bit design but have

[2]The choice of memory technology is orthogonal to the decisions made by the ART algorithm and incurs no security overhead.

great value for larger designs. In other words, our constraints for the ART algorithm would be different for a 32-bit design than they would be for a corresponding 8-bit design. The third design is a normal, unprotected implementation.

Clock frequency remains stable (within 1%) across the three designs. Area and power increase by more as a percentage, about 7% from the insecure design to the secure design and another 8% for the fully scalable design. These area and power increases are small because we are in the low-power domain of microwatts. Since our techniques scale sub-linearly, we expect this overhead as a percentage to diminish dramatically for large SoC designs, which have power budgets on the order of tens of watts.

We lastly provide a breakdown of different modules on-chip in terms of the costs of securing them. This data is shown in Figure 9.12(B). The highlighted components are those used for security purposes, namely reset circuits, reordering circuits, encryption circuits, and the LFSR used for random number generation.

Our experience with TμC1 provided evidence for us that using ART and our implementations of hardware defensive circuits allows us to design and synthesize hardened devices. We learned that the difficulties in applying backdoor defenses boil down to simply understanding the design. So long as the security engineer understands the original design, the additions are easy to apply.

We conducted manual analysis of the interfaces of some publicly available designs from OpenCores (www.opencores.org) – an online database of hardware designs – and found the same techniques we applied to TμC1 also apply to those designs. Additionally, prior work showed through manual analysis that most of the components of an OpenSPARC T2 microprocessor (including those that might be acquired as third-party IP) can be defended with a subset of the techniques in this thesis [Waksman and Sethumadhavan, 2011]. Thus, ART could be applied to that microprocessor, and the additional understanding of the engineering trade-offs would only mean a more efficient application of the defenses.

Perhaps the most important lesson learned from our work on TμC1 is that while the set of all possible circuits is large, most third-party IP fits into a much smaller set of common, reusable circuits. This makes for common interfaces and makes the application of defenses simple and extensible.

9.2 Security Guarantees and Properties of Trigger Obfuscation

The security guarantees provided by trigger obfuscation are relatively straight-forward. We discuss these guarantees and relevant limitations.

Regarding power resets, the method is guaranteed to be secure as long as there is a clear understanding (*i.e.* specification or formalization) of what is considered architectural state. This architectural state can be thought of as any and all state necessary for correct execution according to the ISA. In a system with power resets, microarchitectural state is wiped, while architectural state is preserved. Therefore, if an attacker wants to persist state, he or she is forced to use architectural state. Since architectural state is exactly the same state that should be tested during validation for at least one full epoch, that state cannot contain incorrect values during that testing epoch. However, if there is misunderstanding in practice between the security engineers and the validation team regarding what state is *architectural*, that could lead to problems. For example, if a register goes untested because the validation team does not know about it, then a trigger value could be hidden in that register. Therefore, there needs to be an agreed upon set of state that is architectural, which is allowed to persist and which is validated.

With regards to sequence reordering, whenever sequences can be arbitrarily reordered, the system is guaranteed to be secure with extremely high probability. The odds of guessing a sequence chosen at random is factorially small, which is even smaller than exponentially small (this is because the number of permutations of a finite set of elements grows asymptotically faster than an exponential curve). However, if the sequence is constrained, for example by a memory ordering model, then the insertions of no-ops is required. If no-ops are indistinguishable from regular operations, which is often the case, then the system is still secure. If no-ops are distinguishable, then the system is prone to attacks from sophisticated triggers that ignore no-ops. For this reason, it is important to choose no-ops that do actual work, rather than no-ops that do literally nothing. For example, if operations to a pipeline are being reordered, there are a very large number of instructions other than the no-op instruction that can be used as *de facto* no-ops.

The security guarantee of data encryption is probabilistic but quite strong. The probability of a failure in each case scales as roughly $1 - \frac{1}{2^n}$ where n is the width of the trigger signal, *i.e.* exponential in the size of the data being obfuscation. In practice, n is fairly large – most units that cannot be exhaustively validated to check for backdoors have 64-bit or larger interfaces. So the probability is negligible in practice. One way that the proposed solutions can fail is if the pseudo-random number generator used for the scrambling remaps a trigger to itself, which is extremely unlikely, or if the source of pseudo-randomness is unreliable.

The strength of trigger obfuscation generally relies on having a trusted source for generating psuedo-random numbers. Recently, researchers have sketched attacks on hardware random number generators, though the attacks have low stealth because they are always on [Becker *et al.*, 2013]. Even a very small amount of validation testing can likely detect these attacks. We, however, do not need true cryptographically strong random number generators and can work with weakened pseudo-random number generators. This is mainly because of how we use the random number generation in hardware. One way for the attacker to defeat our scrambling schemes is to learn about the constants for pseudo-random number generation by choosing the plaintext inputs. For instance, forcing the input to zero can reveal a sequence of outputs produced by the random number generator. To generate the trigger despite scrambling, the malicious hardware unit needs to reverse engineer the random number generator within the malicious hardware unit. For this reason, we perform a case study of reverse engineering costs with regard to hardware backdoors. We discuss these costs in the next section. As we will see, the area and power overheads of reverse engineering circuits are too large in practice, which make it reasonable for us to trust our sources of pseudo-randomness.

9.3 A Case Study of Engineering Self-Attacking Hardware

We can consider it important to view any defensive system from the perspective of the attackers. In this section, we endeavor to attack trigger obfuscation from the position of a well-funded adversary. In our model, the attacker has complete knowledge of our defenses and the ability to insert malicious modules into the design. For this case study, we use a cryptographic accelerator (specifically AES). We consider this to be an ideal target for a case study, not only because of the complexity of applying trigger obfuscation to AES but also because of the high level of value and vulnerability in a cryptographic target.

In this setting, we have the following entities:

• **Cryptographic Module Under Test:** This is the untrusted design of a cryptographic accelerator. Our methodology applies to any standard cryptographic system, but our running example is AES-128.

• **Defensive Module:** This is the small trusted code base (TCB), which we call a ***scrambler*** circuit for the purposes of this case study. This scrambler implements a version trigger obfuscation specifically designed for AES. Depending on the exact setting, it can either be assumed trustworthy or formally verified.

• **Attack Module:** We are interested in characterizing attempts to thwart our defensive module and force triggers through to backdoors in the original AES module. We call this last component the ***breaker*** circuit. While the attack module is malicious, we refer to it only as the attack module and not the backdoor. The backdoor is the first-order payload in the cryptographic module. This attack module is a second-order payload, whose only purpose is to cause the original (first-order) backdoor to become triggered.

During the discussion of this case study, it becomes relevant to have a notion of optimality. If we simply implement a poor quality attack, then we learn nothing about our defenses. While creating a completely optimal attack is infeasible, we will consider our attacker's implementation to be *nearly-optimal* if they are algorithmically optimal and implemented using state-of-the-art practices. In other words, while minor implementation tweaks are always possible, and while design libraries and process technologies might change, as long as the attack is essentially optimal for the target technology generation, we can learn from

it. In order for our experiments to be as controlled as possible, we use standard design methodologies and design libraries for each of the components (both offensive and defensive). We anticipate that future changes to design libraries and physical gate features will not substantially impact the relative cost comparisons between the offense and the defense.

9.3.1 Security Engineering Process

Our goal in this case study is to not only measure the best possible countermeasures against trigger obfuscation techniques but also to understand the trade-offs and the balance between the offensive and the defensive aspects of security-aware hardware circuits. To achieve this, we develop a method for evaluating the defenses and the attack surface. Our security engineering process includes the following steps after the desired hardware has been specified.

• **Design and Validation** We design the module under test and validate it with a randomized test bench. We use state-of-practice Verilog-PLI validation [Verilog, 1991; **?**] and validate each transaction and error type for a random distribution of 10,000 inputs.

• **Implementation of Defense** We consider the possible choices we have for trigger obfuscation, such as PRNG size and counter size for timing resets, and we make choices that are pragmatic in hardware. We then implement the defenses and re-validate the design. Design re-validation ensures that the changes we are making have not broken the design and are invisible to validation engineers.

• **Near-Optimal Attack Determination** We determine a near-optimal attack, given knowledge of the defense's parameters. In the presented case study, this includes designing a near-optimal circuit for breaking the PRNG used in the defense.

• **Attack Implementation** We implement the attack circuit and re-validate the design.

• **Synthesis and Evaluation** We synthesize the design with all of the components added. Hardware synthesis includes physical design, such as gate choices, wire placement, layout, floor-planning, and routing. Post-synthesis, we evaluate the costs and trade-offs of each of the components, determining to what extent attacks are feasible and to what extent the defenses are successful.

Together, these five steps replace the design and validation stages in the normal hardware

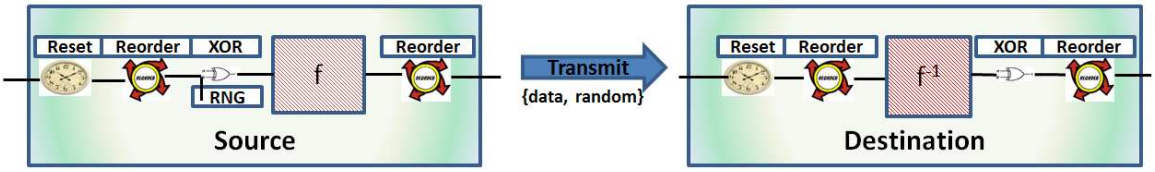

Figure 9.7: An overview of our AES implementation and how it applies both at the source and at the destination of communication. The same hardware unit is used at both source and destination, but the encrypt and decrypt instructions follow slightly different data-paths.

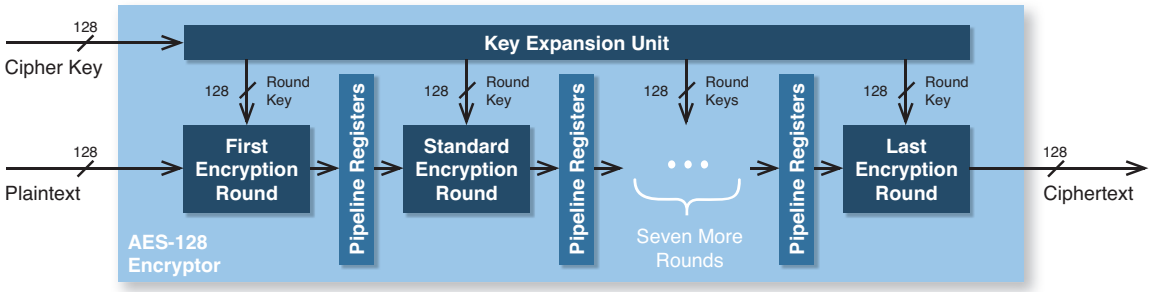

Figure 9.8: The hardware layout of an AES-128 accelerator.

process. An overview of our design is depicted in Figure 9.7. Next, we describe the design details and trade-offs for each of the main components of our system.

AES Unit Design

To emulate the role of a third-party designer, we design a standard AES crypto-accelerator for encryption and decryption. AES accelerators are becoming more prevalent due to the increased performance and security over software implementations. Research groups have pursued ASIC and FPGA designs [Good and Benaissa, 2005; Lu and Tseng, 2002], and AES units have found their way into popular hardware, such as Intel microprocessors [Network, 2012]. Figure 9.8 displays our AES design.

Defensive Scrambler Design

The first defensive technique is power resets. In the case of an AES accelerator, obscuring timing information with resets is straight-forward. Like most slave devices — which peripheral and functional components tend to be — the AES units' outputs are functions

of its inputs and do not take into account long-term state. The state it has only pertains to the transactions it has in-flight. Thus it is acceptable to reset the unit and all memory within the unit periodically after completing in-flight transactions. The implementation for this is little more than a modular counter.

The second defensive technique is sequence reordering. Like many slave devices, our AES unit operates on transactions and has no inter-transaction dependence. Our implementation includes two small circuits. The first is a *deordering* unit at the inputs, which randomly reorders transactions and gives them a local ID. The second circuit is a *reordering* unit, which is essentially a priority queue that uses the ID to reorder transactions.

The final defensive technique is data obfuscation. We chose a crypto-accelerator for our case study because we anticipated it to be one of the hardest cases for practical data obfuscation. Unlike most microprocessor components, which operate largely based on control inputs and mostly move around large data values atomically, cryptoaccelerators are completely dependent on data inputs (key and plaintext) and perform complex operations with them.

The goal of data obfuscation is — at a high level — to apply a homomorphic function (or a hardware analog of this) to the plaintext to allow for encryption. As we saw previously, a simple XOR operation works for most common hardware modules. However, it does not work for all crypto-accelerators, so we must do something more. We leverage the resources we have to perform a similar operation that gives us the same amount of security at a much lower cost. Our algorithm is the following:

Encrypt:

1. Generate a random value R.
2. Perform an XOR of R with the plaintext.
3. Supply the obfuscated plaintext to the AES unit to perform encryption.
4. Transmit both R and the encrypted data (ciphertext).

Decrypt:

5. After transmission, AES decrypt to get back the randomized plaintext.
6. Perform an XOR to get back the original plaintext.

In the original version of the algorithm, there would be two XOR obfuscation phases. Instead, here there is only one such phase. By making this change, we are able to use XOR as our obfuscation function, even though XOR is not truly homomorphic across AES cryptographic operations. This is a highly efficient solution, as it adds only one gate to the critical paths.

This change slightly alters the way the communication is most commonly implemented. The value of R is now entangled in the result and is required now as input to the decryption. This change results in two small but relevant overheads:

1. The result is (in the naive case) up to twice as many bits, as it includes both the ciphertext and R for each 128-bit transaction. However, this can optimized by sending only the initial PRNG seed R for a given encryption task. In the optimized case, the overhead is a constant 128 bits of transmission per document.

2. After decryption, the destination user has two values, the obfuscated plaintext and the publicly known R. For the primary use-case, which is two security conscious parties, both using the same ISA (such as Intel x86 if this were to be incorporated into their AES instruction), this simply requires that the AES instruction in the ISA is specified to take both pieces of data. The AES unit we built, which acts as both an encryptor and a decryptor, has an XOR gate for doing this.

3. Both the source and destination must be aware of the new protocol. For example, in some AES modes, the order of transactions has to be the same at the source and destination. Thus, the decryptor must use the random value R to de-randomize the transaction order after decryption.

Figure 9.9 displays an overview of our implementation. Notice that we are not changing any of the secure channel communications or the functionality of AES. We are only adding public information across the network to be used for scrambling and de-scrambling.

Note that there are two potential types of attackers we are concerned with. The first is a man-in-the-middle attacker who is snooping network packets. The second is a malicious hardware designer who places a backdoor in the hardware prior to fabrication. Note that the former attacker operates at runtime while the latter attacker operates before device fabrication (potentially years prior).

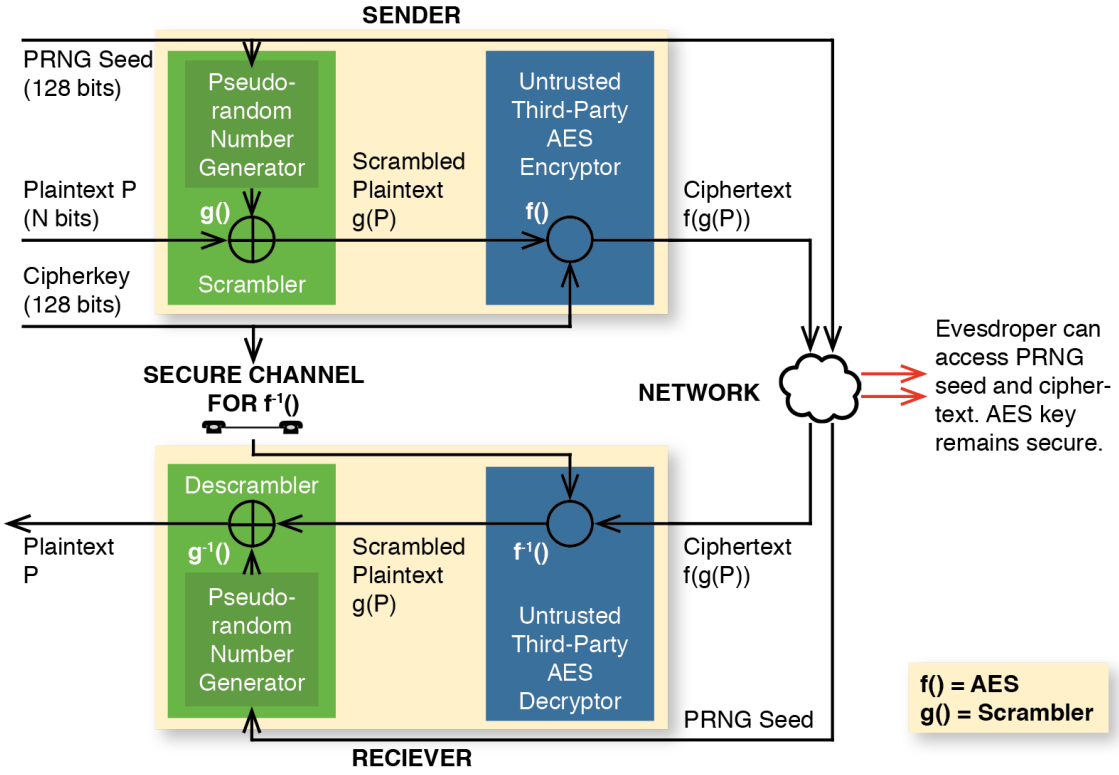

Figure 9.9: An overview of our implementation for secure AES.

The security guarantee in our system comes from the clear demarcation between these two types of attackers. A man-in-the-middle attacker operates on the software/network level and wants access to the secret information that is already readily available in hardware, *i.e.* plaintext or keys. However, it is the attacker's inability to communicate with the internal circuitry of the hardware that thwarts attacks.

The only information that is exposed (*i.e.* sent across the network) is the ciphertext and the PRNG seed R. Note that this value R is public information and can be snooped by the man-in-the-middle attacker without posing any issue. The value of R would only be dangerous if it were known to the internal circuitry of a compromised AES accelerator, *i.e.* if it were known prior to fabrication by the malicious hardware designer.

For the man-in-the-middle attacker who snooped on the network to provide that value R directly to the hardware would be impossible, since that would be a cheat code attack. Consider for example that a user tried to provide R as plaintext to a compromised hardware

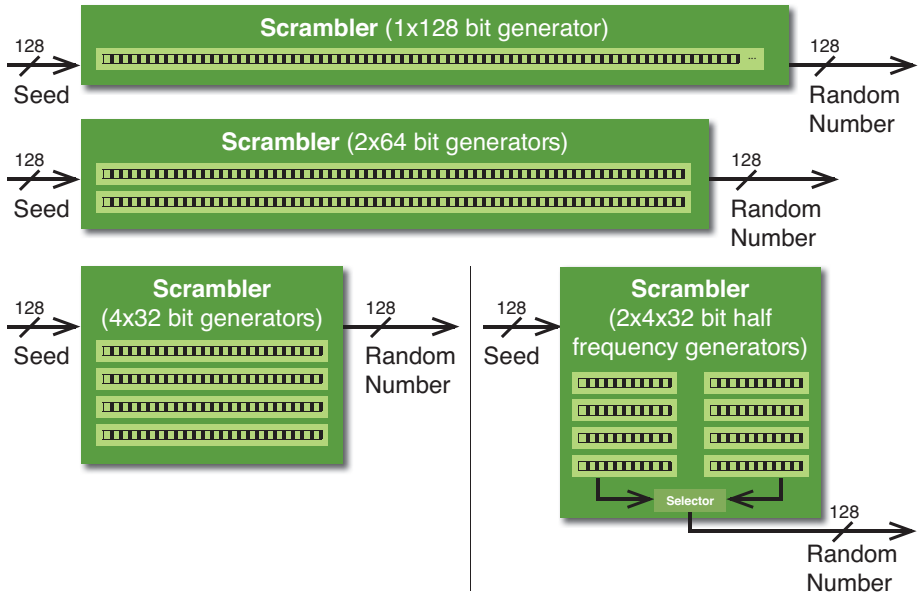

Figure 9.10: The practical options for implementing PRNGs that can produce 128 random bits per clock cycle.

module. As a data input, that value R would be re-encrypted by that later clock cycle's new random value R‘, rendering it useless.

For this reason, the security of this type of system boils down simply to the ability of the hardware's internal logic to produce random numbers that are unpredictable with respect to the malicious trigger logic. This setting is radically different from standard cryptographic settings where packets sent across a network are subject to intense attacks from supercomputers. Instead, we are only concerned with the amount of work that a small, hidden circuit can do in a single clock cycle. At the same time, we are also interested in the quality of the randomness that we can achieve each cycle with a hardware PRNG, which we discuss next.

With regards to selecting a suitable PRNG algorithm for the purposes of this case study, essentially any RNG could be used in our methodology. Hypothetically, a true RNG would make the implementation unassailable because each random value would be a true one-time pad. For this case study, we choose to use pseudo-RNGs (PRNGs) because they are practical. We can also evaluate their implementation costs precisely, as well as the costs of

attacking them.

Many PRNG algorithms exist, including linear congruential generators (LCGs) [Knuth, 1981], multiply-with-carry generators [Marsaglia, 2003], generalized feedback shift register generators [Aiello *et al.*, 1989], minimal standard generators [Park and Miller, 1988], and Mersenne Twister generators [Matsumoto and Nishimura, 1998]. We want a simple PRNG that can match the high clock speed of the AES unit without extreme pipelining and without significant area cost. We also want our PRNG to have mathematical properties that will allow us to understand the limitations of our attacker.

For the purposes of our evaluation and implementation, we choose low bit-width LCGs, as these are efficient enough for immediate real-world applications. Some options for the parameters of the LCG are depicted in Figure 9.10. LCGs are a class of PRNGs that have a generating equation of the form:

$$X_n \equiv (aX_{(n-1)} + c) \pmod{m}$$

where a, c, and m are the constants that define the particular LCG. Because the general modulus function is computationally intensive, the value of m is usually chosen to be the word size (a power of 2) of the underlying architecture, so that performing the modulus in hardware is trivial. We consider designs that use 2^{32}, 2^{64}, and 2^{128} as m.

The constants a and c are selected to yield a long number sequence period using recommendations from Knuth [Knuth, 1981]. We summarize them here. When m is a power of 2, a should be chosen so that $a \equiv 8 \pmod 5$ and should not be close in value to either 0 or m. The constant c should be relatively prime to m, which means it should be an odd number. Lastly, a and c should both be greater than $\frac{m}{2}$ to prevent the use of direct division operations. Applying these restrictions on the values of the constants a and c prevents trivially attackable cases from occurring and prevents attackers from leveraging direct divides, bitwise operations, or simple bit-shifting schemes, as methods for quick attacks. Instead the attacker is forced to compute the actual inverse value, which is a non-trivial operation [Knuth, 1981] and which we will see in our evaluation is infeasible. However, in Section 9.3.2 we will also empirically analyze the potential for performing probabilistic attacks in hardware.

Thus, it suffices to restrict a such that $a \equiv 8 (\text{mod } 5)$ and $\frac{m}{2} \leq a \leq \frac{3m}{4}$ and to restrict c such that $c \equiv 1 (\text{mod } 2)$ and $\frac{m}{2} \leq c$. Thus for m with n bits, the number of a values is $\frac{2^n}{8*4} = 2^{n-5}$ and the number of c values is $\frac{2^n}{2*2} = 2^{n-2}$. Therefore, the space has size $n^{n-5+n-2} = n^{2n-7} > 2^n$ for reasonable values of n, making the space of choices exponentially large and thus not subject to exhaustive search attacks.

With the PRNG algorithm determined, the scrambler unit must meet two design requirements:

1. The generator hardware must match the clock speed of the AES unit, generating a new random number for each plaintext input

2. The random number must be 128-bits long to match the plaintext word size.

Requirement 1 can conceivably be relaxed: we can opt to generate a new random number once for several cycles of plaintext input. We choose not to relax this requirement because using the same random number for multiple inputs makes it easier for an attacker to compromise security.

An LCG requires a k-bit multiplication, where k is the width of the generated numbers. Since a 128-bit multiply would require pipelining and relatively high area costs to match clock frequency requirements, we choose to generate the 128 bits piecemeal from multiple small LCGs. Furthermore, the designer could choose to alternate the output of two or more PRNGs to get lower frequency.

Multiple choices of how to break up the generation of these 128 bits are viable, and one of the parameters we explore is how to most efficiently do this. We discuss the results of implementing these options in Section 9.3.2.

Offensive Breaker Design

The purpose of the breaker module is to design a near-optimal circuit that the attacker can embed in the AES IP to circumvent the scrambler. The attacker knows exactly what our defense circuit is and knows the optimal algorithm for attacking it. The attacker's goal is to obtain knowledge of the number sequence being generated by breaking the PRNG algorithm. The attacker can leverage this knowledge to inject plaintext that, after predictable scrambling, becomes a backdoor trigger.

The breaker infers the PRNG output by observing the scrambled form of some known plaintext, supplied by the attacker or an accomplice. Given three consecutive outputs of the PRNG, the attacker can discover the hidden constants mathematically. The solution requires at its core finding the inverse of a number in a modular space. This is a well studied problem from the field of cryptography. The most efficient solution (for non-trivial choice of constants) is the extended Euclidean algorithm [Liu *et al.*, 2008].

The extended Euclidean algorithm is an iterative algorithm, which requires $\mathcal{O}(log(N))$ iterations, where N is the size of the modular base. Each of the iterations is essentially equivalent to a division that reduces the magnitude of the inputs, hence the logarithmic performance. Part of our focus will be spent on incorporating a hardware divider efficiently.

Given that the attacker knows the value of m, which is simply a power of two, and has recorded three straight inputs — x, y, and z — from known plaintext, the breaker circuit solves the following algebraic problem. From the generating equation of the PRNG:

$$y \equiv ax + c \pmod{m} \tag{9.1}$$

$$z \equiv ay + c \pmod{m} \tag{9.2}$$

Subtracting the two equations we obtain:

$$z - y \equiv a(y - x) \pmod{m} \tag{9.3}$$

Multiplying equation 9.1 by $(y - x)$ yields:

$$y(y - x) \equiv xa(y - x) + c(y - x) \pmod{m} \tag{9.4}$$

Substituting for $a(y - x)$ using equation 9.3 yields:

$$y^2 - yx \equiv xz - yx + c(y - x) \pmod{m} \tag{9.5}$$

Finally, cancelling the yx terms and rearranging terms yields:

$$y^2 - xz \equiv c(y - x) \pmod{m} \tag{9.6}$$

Solving equation 9.3 and equation 9.6 in modular space to isolate a and c requires finding the modular inverse of $(y - x)$. The extended Euclidean algorithm solves for equations of the form

$$pu \equiv 1 \pmod{q} \tag{9.7}$$

Here p and q are coprime, and u is the variable to be solved for. With $(y - x)$ as p and m as q, we find the modular inverse of $(y - x)$ by solving for u.

The number of iterations required for the extended Euclidean algorithm to succeed depends on input values. In hardware, we implement a module that carries out one round of the extended Euclidean algorithm. This module can be chained together to increase the probability of success. The goal of an attacker with limited resources is to find the optimal trade-off between probability of success and cost. We discuss implementation trade-offs for the breaker and the scrambler in Section 9.3.2. We will see that it is not feasible to implement the full Euclidean algorithm, but it is possible to achieve a small probability of success with only one iteration.

9.3.2 Evaluation of Offensive and Defensive Costs

We next evaluate trade-offs for the design parameters described in Section 9.3.1. We focus on the performance of the chosen designs, primarily clock frequency and design area, for each of the components in our design. Our results show that it is infeasible for the attacker to make a deterministic attack against trigger obfuscation techniques.

The cost of a deterministic attack is high because it takes a deep pipeline to guarantee a correct prediction. As the pipeline gets shallower, the probability of a correct guess shrinks, but not all the way to zero. Nevertheless, this opens up the possibility for an attack that has no guarantee of success but has a better chance of supplying the trigger than one would have with random guessing (which would be $\frac{1}{2^{128}}$. We analyze the trade-offs between success probability and design feasibility for the attacker to see if there is a design point that presents a significant threat.

We implement the modules in our design using standard ASIC cells and 90nm libraries. We synthesize these designs using the Synopsys Design Compiler [Synopsys, 2006] and scale area results to 45nm, a common node for embedded systems, using ITRS conversions [ITR,]. We note that custom circuits can be optimized beyond that of which ASIC design is capable. Our goal is to understand the relative costs between defenses and attacks. If both

Table 9.5: Synthesis Results of PRNG Options. Area overhead is shown as a percentage of the original design area.

Width	Depth	Frequency	Area Overhead
1x128 bits	2 Stages	260 MHz	22%
2x64 bits	2 Stages	257 MHz	12%
4x32 bits	2 Stages	244 MHz	6%
2x(1x128) bits	2 Stages	142 MHz	36%
2x(2x64) bits	2 Stages	158 MHz	20%
2x(4x32) bits	2 Stages	153 MHz	10%

attack and defense were to be done custom instead of in ASICs, we believe the comparative results would be very similar. We next present the cost breakdown for each module.

9.3.2.1 Baseline AES Module

Our pipelined AES unit achieves 238 MHz operating speed with a throughput of one encryption per cycle. This is comparable to prior similar efforts [Good and Benaissa, 2005]. The area cost is about 0.625 square millimeters.

9.3.2.2 Defensive Scrambler Module

As described in Section 9.3.1, several configurations for the PRNG are viable. The designer can opt to generate all 128 bits of the random number at once, thereby accepting the penalty of having to implement a 128-bit multiplier. On the other end of the spectrum, the designer may generate four 32-bit random numbers in parallel using less expensive 32-bit multipliers. Additionally, the designer can trade off area and frequency by using twice as many generators, each running at half frequency. We tried all practical configurations, including all reasonable pipeline depths. The best designs are shown in Table 9.5. Full frequency generators (the first three entries) have to reach at least 238 MHz to be viable, while half frequency generators (the last three entries) have to reach at least 119 MHz. For our purposes, the most practical choice is the full frequency, 4x32-bit PRNG. The results of this study clearly demonstrate the practicality, both in terms of area and frequency, of

Table 9.6: Synthesis Results of a Euclidean Algorithm Stage (Breaker Circuit)

Divider Depth	Multiplier Depth	Clock Frequency	Area Overhead
11 Stages	3 Stages	252 MHz	22%

using LCGs to implement data obfuscation.

9.3.2.3 Breaker Module Trade-Offs

Each iteration of the extended Euclidean algorithm requires a division and a few multiplications. Without pipelining, 32-bit dividers and multipliers do not keep pace with the AES unit's 238 MHz clock. The minimal amount of pipelining required to meet the timing goal is eleven stages for the divider pipeline and three stages for the multiplier. Exact numbers are shown in Table 9.6.

In order to turn a single stage, which represents one iteration of the algorithm, into a full solver, an attacker has two broad microarchitectural choices. The first choice is to tile many (about 50) stages and pipeline them. The second choice is to make many duplicates, have them feed back on themselves, and time multiplex the work across all of them. In either case, this involves tiling about 50 copies of the circuit. Our results show that neither of these solutions is feasible because even a single stage requires a significant area overhead.

These results completely rule out the possibility of deterministic attacks. Therefore, from here on out we concern ourselves with probabilistic attacks. Probabilistic attacks are potentially more promising because after only one or a few iterations of the Euclidean algorithm, there is a non-zero chance of finding the solution, due to the fact that the algorithm sometimes finishes faster than the worst case number of iterations. Next, we discuss what degree of success can be achieved in hardware.

9.3.2.4 Euclidean Algorithm Stage Count Evaluation

We run the extended Euclidean algorithm many times in software to empirically determine its probability of success for discovering the PRNG constants for our setting after limited iterations. Figure 9.11(A) shows the overall results, and Figure 9.11(B) shows a close-up of

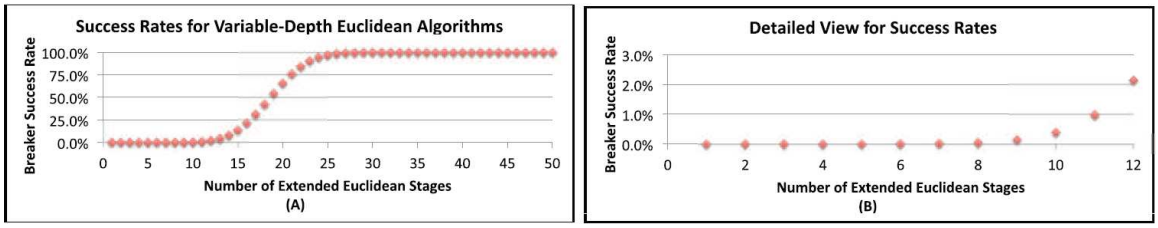

Figure 9.11: The expected success rates of a breaker unit against our PRNG as a function of the number of iterations of the extended Euclidean algorithm.

Table 9.7: Synthesis Results of Breaker Unit

Probability of Success	Breaker Depth	Area Overhead
1 in 1 Billion	1 Stage	22%
1 in 1 Million	4 Stages	45%
1 in 10 Thousand	7 Stages	112%
1 in 100	11 Stages	210%
1 in 10	15 Stages	291%
1 in 2	19 Stages	415%

low probability cases.

As Table 9.7 shows, with the minimal choice of a single stage, an attacker still has roughly a one in a billion chance of success, which is low, but not too low to be a threat. For example, if our AES unit is running at full capacity, it can perform a billion operations in just over four seconds. The degree of threat this poses depends on the scenario. If an attacker is in a position to supply a backdoor trigger for four seconds (or more) then this is a very real threat. In other scenarios it may not be as much of a threat.

Having the degree of the threat quantified can inform security operatives in the field. If this probability of attack is acceptable, then it can be ignored. If this probability is too high, then they know the design is insufficient. A few options are available for paying to strengthen the defense. An ideal (if practical) fix would be to use a true (physical) RNG. If a true RNG can be efficiently made with 128 bits-per-cycle throughput at high frequency, then trigger obfuscation can be made *completely* secure, as demonstrated by our analysis. Another more immediate possibility would be to use a 64-bit LCG instead of a 32-bit LCG.

If a defender is willing to pay the extra area cost for that, then the threat, though not erased entirely, would be diminished beyond reason, because the probability of success would be orders of magnitude less.

Figure 9.12(A) goes into more detail regarding the area costs required to achieve varying success rates. As an adversarial designer tunes the parameters to increase success rate, the area cost grows very quickly. There's a sweet spot in the middle of the curve where an attacker can get a lot of increased success for little marginal cost. However, to reach that part of the curve requires more than a 200% area overhead. An interesting question for the future is: how much area is too much? Increasing the area and code base of a design by several fold would be noticed and most likely make the unit too expensive to market. However, it is not known where the line is or whether 22% is too much. It is likely that 22% would be far too much to evade IC fingerprinting [**?**] or similar techniques, but more work is needed in those areas to know for sure. Figure 9.12(B) displays the respective area costs of each of the components assuming the minimal breaker circuit.

9.3.2.5 Design and Code Complexity

While code complexity is a function of the person coding, we use source lines of code as a first order approximation of the coding complexities of the different modules in our design. Table 9.8 shows the relative code sizes of these modules. All code is structural, synthesizable Verilog. The original AES unit comprises roughly five thousand lines of code. It is possible to build an AES unit with fewer lines, but our goal of optimizing for area resulted in this code size, which is reasonable for real world applications.

The adversarial circuit comprises roughly a 10% code bloat. It is unclear how much code bloat is noticeable in practice, and this is an interesting subject for future discussion. The defensive circuits together comprise a TCB of roughly 3.3%.

9.3.2.6 Generality of Case Study and Extensions

Our method for protecting AES works because AES consists of two steps that perform inverse operations. Since the data before AES encryption and after AES decryption is the same, this is a good use-case for what we call *psuedo-homomorphic encryption*. This also

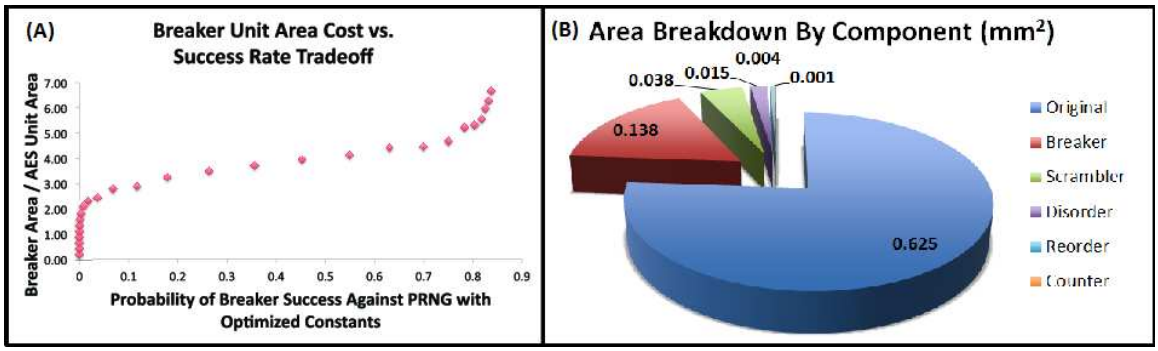

Figure 9.12: (A) This curve shows the risk of a malicious designer's counter-attack being discovered. If a 10% probability is acceptable to the designer, then the defender should not notice a three-fold area bloat. On the other hand, given a fixed area budget, an attacker can only achieve a bounded probability of success. (B) Area costs of each component at 45nm, using the smallest possible breaker circuit, *i.e.* the one with the smallest non-zero chance of success.

works in similar fashion for any standard cryptographic algorithm, because encryption and decryption have this property of inversion (limitations for AES and other common crypto algorithms are discussed in Section 9.3.3). A cryptographic unit applies an encryption e along with a decryption d under the requirement that

$$d(e(x)) = x \; \forall x$$

Our requirement is similar but with the addition of the XOR operation.

$$R \oplus d(e(R \oplus x)) = x \; \forall x$$

The two requirements are equivalent, so our requirement does not limit the set of cryptographic algorithms that can be used. In fact, our techniques can be applied to any function and its inverse, where f and f^{-1} play the role of e and d. For this reason our circuits can be applied directly to a wider class of hardware modules because many modules, such as buffers, interconnects, memories, caches, comparators, reorder stages, fetch units, etc. move around and store data items without changing them arithmetically. As we discuss in Section **??**, microarchitectural studies are needed to determine the costs and benefits of such applications.

Table 9.8: Relative Coding Complexities of Design Components

Module	Submodules	Lines of Code
AES Unit	44	4976
Breaker	5	506
Scrambler	4	99
Deorder	1	39
Reorder	1	31
Counter	1	14

9.3.3 Future Attacks: Hybrid Hardware-Software Backdoors

We briefly discuss a topic that we have not studied in depth and that we consider interesting as an avenue for future research. Hardware backdoor research to date (including our research) has assumed that hardware attacks and software attacks are distinct. While hardware backdoors might be triggered by malicious software, we assume that the software is performing actions that would be harmless without the malicious hardware underneath.

However, consider the possibility that a hardware backdoor is used to exacerbate the damage of an existing software attack. There is no reason this should be impossible. While there are several possibilities, we consider this option as it pertains to our cryptographic design. With options for pure hardware attacks diminished by our defenses, an intelligent attacker might turn to a hybrid hardware-software attack. As an example of this, we believe that the change in protocol necessary for the our hardware-oriented defense of cryptographic hardware opens the door for hybrid attacks. We briefly discuss two such attacks in the context of this work while acknowledging that other attacks might be possible.

The first attack we propose involves an active man-in-the-middle. Suppose there exists a man-in-the-middle on the network who can both intercept messages and alter them at will. This attacker has already succeeded at attacking the system at the software and network level. However, the damage of this attack can be exacerbated using a hardware backdoor. This attacker has the ability already to replace real ciphertext with fake ciphertext before it is received. This is sufficient to perform a denial-of-service attack, as well as possibly an

integrity (or replay) attack. When conspiring with a hardware-based attacker, this attack could be expanded to a confidentiality attack. The attacker could replace the original ciphertext with spurious ciphertext that corresponds to a hardware cheat code. The cheat code could then cause hardware to release the key, breaking confidentiality.

We note that this type of attack is likely to be an emitter backdoor and could potentially be detected by **TrustNet** or a similar system. However, there is no reason to necessarily assume that a company that has invested in trigger obfuscation will also invest in **TrustNet**. Currently, we consider this type of hybrid exacerbation attack to be an open problem for future research.

A second type of attack that we propose could occur on a shared system. Suppose that a benign user and a malicious user share a software system and that each user has his or her own AES key. The key is not obfuscated because it is needed for computation and because it is by definition secure (if the attacker knows the key *a priori* then the game is over). However, suppose that the malicious user supplies a cheat code via a pre-chosen key that is meant to cause a permanent operational change, which would remain active when the benign user supplies the real key at a later time. This approach does not work on our implementation, because we reset the device on a key change. However, from a practical security standpoint, it is important to ensure that reset is implemented in this fashion, and it is not unthinkable that other implementations might be configured differently. There exist ways to verify that state-holding elements are resettable, such as code inspection or verification of a small amount of code, but these methods are not always employed.

In general, the idea of using hardware backdoors indirectly as a tool for aiding malicious software is an interesting area and is a natural direction for researchers to pursue in the future.

9.4 Conclusions and Future Directions for Trigger Obfuscation

At present, trigger obfuscation is an intriguing and powerful technique that still presents several open questions. Each of three components of trigger obfuscation will continue to raise interesting questions as the space of hardware circuits is explored further.

Rapid resets may need to be adapted for non-volatile logic, including the possibility of increased usage of non-volatile memories (such as using Phase Change Memory). Rapid resets also might need to be adapted for stateful accelerators, programmable accelerators, or FPGA usage.

Sequence reordering raises questions of its own. In addition to the increased variety of accelerators being used, some of which are transactional and some of which are not, increasingly complex memory systems and on-chip routers make it important to understand when sequence matters and when it does not.

Data obfuscation is perhaps the most open of the three methods. Homomorphic obfuscation is a new area that is distinct in some respects from older work on fully homomorphic encryption. Constructing a library of homomorphic circuits or generalizing techniques for domain-specific homomorphic circuits could have value to the community. Additionally, a further look into homomorphic cryptographic circuits would have value, as cryptography is an area where homomorphic circuits are both important and especially hard to implement.

Lastly, hybrid hardware/software attacks raise questions for future study. The work in this field to date has assumed either no interaction or relatively simple interaction between software and the hardware backdoor circuits. However, if sophisticated software attacks are combined with hardware-oriented attacks, the degree of compromise needed by the hardware could be significantly decreased and pave the way for more subtle attacks. We believe this area deserves further inspection in the future.

Chapter 10

Detecting and Reacting to Backdoor Payloads Dynamically at Runtime

10.1 Backdoor Payload Detection Overview

If a backdoor is somehow designed to bypass both static analysis and trigger obfuscation, thus allowing itself to be turned on at a maliciously intended time, then we are forced to detect the payload of the attack at runtime. This is the last of the three steps in our defense-in-depth design-side security approach.

When detecting a malicious payload, we must either recover the system to a coherent state or fail gracefully. This ensures that in the worst case, exfiltration of system data (and thus the compromising of system privacy) cannot occur. To make this happen, we use a dynamic on-chip monitoring system that performs backdoor payload detection by continuously checking microarchitectural invariants.

Key Insight: Our system for on-chip monitoring is based on two key observations: 1) that in a hardware system, the communication events that exist between multiple hardware units are largely deterministic, and there exist invariants that are violated only when backdoors achieve their payloads, and 2) that the way hardware modules are connected to each

other lends itself naturally to a self-monitoring system. Examples of invariants between hardware units can be simple checks such as that the number of instructions executed by the functional units should not be larger than the number instructions that are processed by the instruction decoder. Our second observation points out that if we make the weak and reasonable assumption that the majority of modules are not malicious, we can build a secure invariant-monitoring infrastructure even if we do not know which of the modules are the malicious ones.

For our initial defensive implementations, we assume that exactly one module (or one design team) is malicious, but there are natural extensions to n malicious modules for $n > 1$ [Waksman and Sethumadhavan, 2010], which we discuss later. Increasing the fraction of the design that is malicious naturally increases the cost of the self-monitoring network.

10.2 The TrustNet Defense System

The first self-monitoring system we build is called **TrustNet**. The basic idea, depicted in Figure 10.1, is to have self-monitoring triangles formed out of any set of three hardware modules. In typical hardware designs, any module is connected to several others. For example, in a microprocessor pipeline, a decoder is connected to a fetch and an execution unit, among others. There is usually some unit that sees data before it (in this case the fetch) and some unit that sees data after it (in this case the execution). We call these the *predictor* and the *reactor* respectively. By adding a verification step between the predictor and the reactor, we can confirm that the untrusted (monitored) unit is doing reasonable work. For example, the fetch unit and execution unit can communicate to ensure that the correct number of instructions are being decoded. This simple example invariant ($instructions\quad in = instructions\quad out$) prevents all decoder-based code-injection attacks, an important class of attacks.

Using **TrustNet**, we can protect against any emitter backdoors. Recall that by definition emitter backdoors cause the wrong number (either too many or too few) of microarchitectural transactions to occur in the compromised unit. **TrustNet** counts these events and matches them up on a cycle by cycle basis. Thus, as soon as one too many or one too

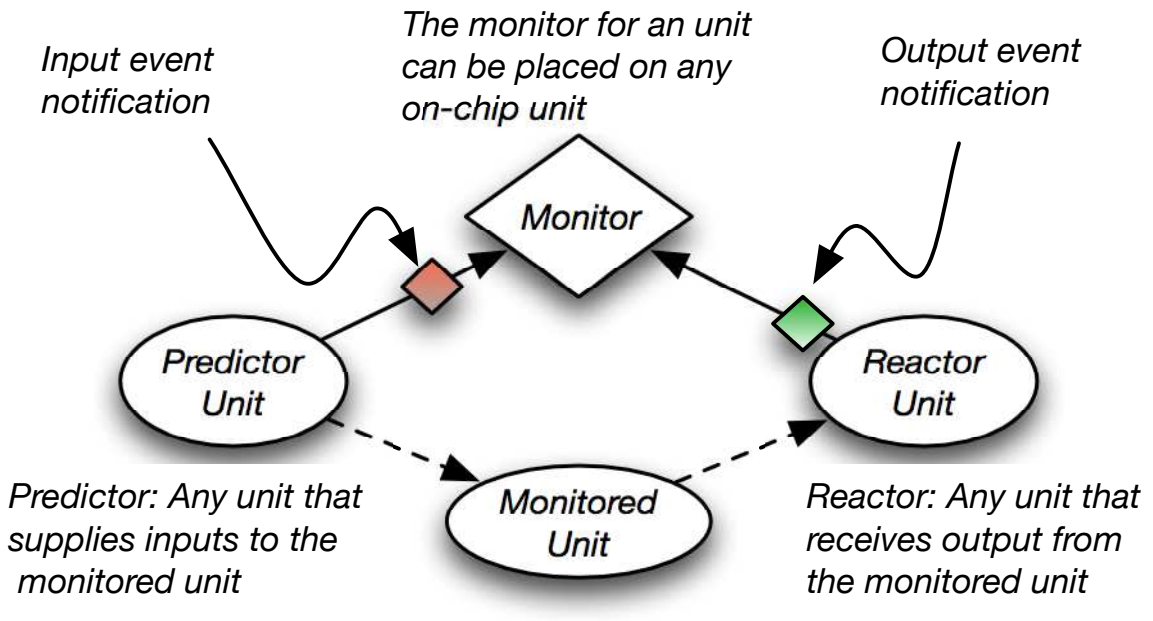

Figure 10.1: Overview of the **TrustNet** and **DataWatch** monitoring scheme. The triangular microarchitecture provides the necessary distribution of work in a simple and easy-to-implement fashion.

few communications occurs, **TrustNet** detects this misbehavior. Conceptually, the system detects violations of deterministic communication invariants between on-chip units, which are the same invariants necessarily violated by emitter backdoors.

The microarchitecture for a single monitoring unit is a prediction/reaction monitor triangle, depicted in Figure 10.1. A triangle consists of three different on-chip units - a predictor, a reactor, and a target (monitored unit in Figure 10.1. The predictor unit sends messages to the monitor, predicting events that should come out of the target unit. If the reactor does not receive a predicted event, or if the reactor receives an event that was never predicted, then the monitor throws an alarm.

The target unit is any unit on-chip. For one example, using Figure 10.3 as reference, consider the decode unit (IDU) as a target. The predictor unit can be any unit that sees events before they reach the decode unit, for example the fetch unit (IFU). The fetch unit sees every instruction before it reaches the decode unit. The reactor unit can be any unit that sees events after they pass through the target. For example, it can be the execution

unit (EXU), because that unit always receives information about instructions after they pass through the decode unit. The monitor itself can reside anywhere on-chip. The IFU knows each cycle whether or not it has fetched a new instruction. The EXU knows at each cycle whether or not a valid instruction is advancing through the pipeline. Having these two sources of knowledge corroborate prevents the IDU from ever inserting bogus instructions into the pipeline. Nothing extra has to be computed, since this knowledge is already represented by signals in the respective units.

A vital aspect of this design is that the target unit never communicates with the monitor. Therefore, even if the designer of unit X knows the design of the monitor (which is likely), the designer is unable to corrupt the monitor of X. For this reason, it is important that the monitor of X should not be physically part of unit X.

Recall that in this setting, at most one sub-team is corrupt, and so an attacker cannot corrupt two independent units on-chip. This assumption guarantees that **TrustNet** reliably detects attacks. Consider the monitor set up to watch unit X. There are four items in play - the predictor of X, which we will call P; the reactor to X, which we will call R; X itself; and the monitor of X, which we will call M. The attacker must choose one and only one of these items to corrupt. In order to cause a confidentiality or integrity attack in X, the attacker would have to choose to corrupt X itself. However, in that case P, R, and M are all untampered with, and the attack is caught at runtime by the P-R-M monitor. The attacker could instead tamper with P or with R (but not both). In either case that attack would be caught at runtime, since P and R would disagree with each other during the cycle when the first faulty communication occurred. The last choice is to corrupt M to cause an availability attack (a false positive in the detector). However, since M contains only one (or a few) logic gates, it is too small to contain a cheat code or ticking time bomb. M has so few states (usually none) and input bits (usually two) that it can easily be verified statically by exhaustive enumeration. Thus, in this worst case, only denial-of-service is possible, and even this worst case is impossible in practice due to the simplicity of the microarchitecture.

10.3 The DataWatch Defense System

As the natural extension to **TrustNet** we designed a more robust system called **DataWatch**. This system covers a wider class of attacks at the cost of slightly higher overheads. Since **TrustNet** measures simple invariants that can be checked by counting, such as numbers of instructions or events, **DataWatch** performs simple data integrity checks. For example, **DataWatch** can check that memory events have the correct types or that the correct opcode types are being supplied to pipeline units. **DataWatch** is inherently more heavyweight than **TrustNet**, since **TrustNet** uses one-bit signals and **DataWatch** uses multi-bit signals. However, **DataWatch** is still efficient for modern designs. Thus, while **TrustNet** is an ideal solution for emitter attacks, **DataWatch** is an effective way to also catch corrupter attacks, making the decision of which system to use dependent on the use case.

DataWatch is specifically best at catching control corrupter backdoors. Recall that control corrupters change the semantics of an instruction in flight. The same predictor/reactor scheme, used to catch emitter attacks, works to detect control corrupter attacks, since these attacks either cause reactions (microarchitectural transactions) that are not predicted or prevent transactions that are predicted. **DataWatch** uses the same triangle scheme as **TrustNet**. However, instead of transmitting individual bits to monitors, signatures (*e.g.,* hashes, parity bits, slices) are passed to monitors that check for equivalence. For example, in the case of the decoder, a two bit signature is enough to define whether an instruction involves a load, a store, both, or neither (two bits for four cases). An equality check on the signatures from the predictor and the reactor is enough to confirm an on-chip invariant, namely that the number of each type of regular memory operation that enters the IDU equals the number of each type of memory operation that exit the IDU. This prevents, for instance, an attack that exfiltrates data by converting no ops into store instructions.

10.4 Handling Data Corrupter Attacks

Data corrupter attacks are both the hardest to implement and the hardest to defend against. In practice, they may be infeasible to design, but they are academically interesting and remain the only type of attack that can potentially evade the **DataWatch** defense system.

Table 10.1: Comparison of **TrustNet**, **DataWatch**, and smart duplication for simple, in-order microprocessors

Attribute	**TrustNet**	**DataWatch**	**Smart Duplic.**
Data Corrupter Detection	No	No	Yes
Control Corrupter Detection	No	Yes	Yes
Emitter Detection	Yes	Yes	Yes
False Positives	No	No	No
Stalls Processor	No	No	Possibly
Area Cost	Negligible	Low	Moderate
On-Chip Traffic	Increases		
Off-chip memory	No effect		
Backdoor source	Can be identified		

Data corrupters change only the data that is sent in on-chip communications. These backdoors are fundamentally different from the types previously discussed because the amounts and types of communications between units during the execution of an instruction is identical to that of a correctly functioning chip. The monitor triangle, while very efficient for recognizing amounts and types of transactions, does not work well for this case, because data corrupter attacks cannot be recognized without duplicating some of the computational logic that has been corrupted. For example, if the EXU (execution unit) produces an incor-

rect sum, the fact that the sum is wrong cannot be known without duplicating (or otherwise performing the job of) the ALU (arithmetic/logic unit).

However, this type of attack shares interesting similarities with transient errors that can occur in microprocessors. Significant work has been done toward transient error detection [Chatterjee *et al.*, 2000][Reinhardt and Mukherjee, 2000][Yoo and Franklin, 2008][Carretero *et al.*, 2009] and fault tolerance, and we draw on the principles of some of this prior work. It is sufficient in many cases to duplicate select computational logic in order to protect the RTL design, since standard memory structures (*e.g.,* RAMs) are not susceptible to RTL level attacks. We propose that this type of minimal duplication, which we call 'smart duplication,' can be used in a case-by-case way to protect any units (*e.g.,* memory control unit) that are not covered by the **DataWatch** system or any units that may be considered vulnerable to data corrupter attacks. This partial duplication allows for protection against data corrupter attacks. However, it does this at the possible cost of processor stalls and extra area, and in most domains data corrupter attacks would likely be considered infeasible due to the requisite of knowing the binaries that will be run in the future during the RTL design phase. Therefore, this technique may only be useful in a few select domains or not at all.

Table 10.3 summarizes some of the attributes of the offered solutions. None of the proposed solutions have a problem with false positives (false alarms) because they use invariants that can be easily determined statically in non-speculative, in-order microprocessors. Extending this solution to designs with advanced speculative techniques, such as prefetching, may make false positive avoidance non-trivial. False negatives (missed attacks) are only a problem if multiple signals in the **DataWatch** technique are hashed to save space, because two different values may hash to the same key, thus tricking the equality checker. However, hashing is an implementation option, which we chose to avoid because the space requirement of the baseline **DataWatch** system is fairly low.

10.5 Handling Detection Alarms

We have thus far discussed how **TrustNet** and **DataWatch** can detect attacks. However, if a payload is detected, there remains the important question of what to do with that informaiton. There are several possibilities for techniques for handling alarms from **TrustNet** and **DataWatch**. A simple and legitimate response could be to shutdown the entire system. At the least, this would turn confidentiality or integrity attack into an availability attack, because the only result would be denial of service. In highly secure domains, this may be desirable to guarantee no exfiltration of sensitive data. For example, if an attack is meant to destroy crucial equipment, simply shutting down is likely less catastrophic. If continuous operation is a higher priority, then some form of N-way redundancy is an option. If designs from multiple vendors are available, then when an alarm sounds, the computation can be migrated onto another machine from the machine that has been compromised, with computation being resumed from the last known safe state. As an arbitrary example, if the alarm sounded on an Intel x86 chip, the computation could be moved to an AMD x86 chip and resumed. This proposed technique could also be helpful for forensics. When an alarm goes off, the data in flight in the hardware units could be flagged as a cheat code and logged for future execution.

10.6 Security Guarantees of TrustNet and DataWatch

We briefly discuss the guarantees provided by our proposed runtime systems. Consider a monitor that is set up to watch an untrusted unit X. There are four actors in play – the predictor of X (P), the reactor to X (R), X itself, and the monitor of X, which we call M. With a lone-wolf attacker model, the attacker gets to corrupt one and only one of these actors. In order to compromise confidentiality or integrity, the attacker would have to choose to corrupt X itself. In this case, P, R, and M are all untampered with, and the attack can be detected. If the attacker chooses to tamper with P or R, then they would disagree, and the attack would be detected. The attacker can choose to attack M, but since monitors are one or a few logic gates (XOR gate), it can be formally checked not to contain backdoors. The proposed solution, however, will not work for a conspiracy

between designers of multiple units or when one bad designer designs multiple units that are responsible for monitoring each other. Organizational security is necessary to prevent such attacks.

10.7 Case Study and Evaluation of TrustNet and DataWatch

As a case study, we demonstrate the effectivness of **TrustNet** and **DataWatch** on portions of the Sun Microsystems' OpenSPARC T2 microarchitecture. In this study, we use the HDL hardware implementation of OpenSPARC to systematically determine the number of on-chip units that can be readily covered (without ingenuity or further expansion) by our design. To measure the vulnerability surface we observe that a hardware unit is only vulnerable to backdoors in-so-far as its interfaces are threatened. The processing that goes on inside the unit can be checked to be safe simply by checking its inputs and outputs. Thus the interfaces present points of vulnerability, and the efficacy of our solution is then determined by whether or not these interfaces are protected from attacks using **TrustNet** and **DataWatch**. Figure 10.2 shows the units on an OpenSPARC chip that can be covered partially or fully using **TrustNet** and/or **DataWatch**. We note that this is a conservative estimate, based only on those interfaces we can easily cover. Further investment of effort or domain-specific knowledge would likely lead to even better coverage.

To get into more detail, the in-order microprocessor used in our simulations closely models the cores and cache hierarchy of the aformentioned OpenSPARC T2 microprocessor and is based on manual analysis of the OpenSPARC source code. For this study, the units in the processor core are partitioned as described in the OpenSPARC T2 documentation, and we used the open-source HDL code to identify the predictors and reactors for each unit. The following are the monitoring triangles we implemented in simulation, categorized by the untrusted unit being monitored. We first list the **TrustNet** triangles.

• *#1 IDU:* The primary responsibility of the IDU is to decode instructions. Predicted by the IFU and reacted to by the EXU, the IDU monitor confirms each cycle that a valid instruction comes out of the IDU if and only if a valid instruction entered the IDU. This monitor detects any attack wherein the IDU inserts spurious instructions into the stream.

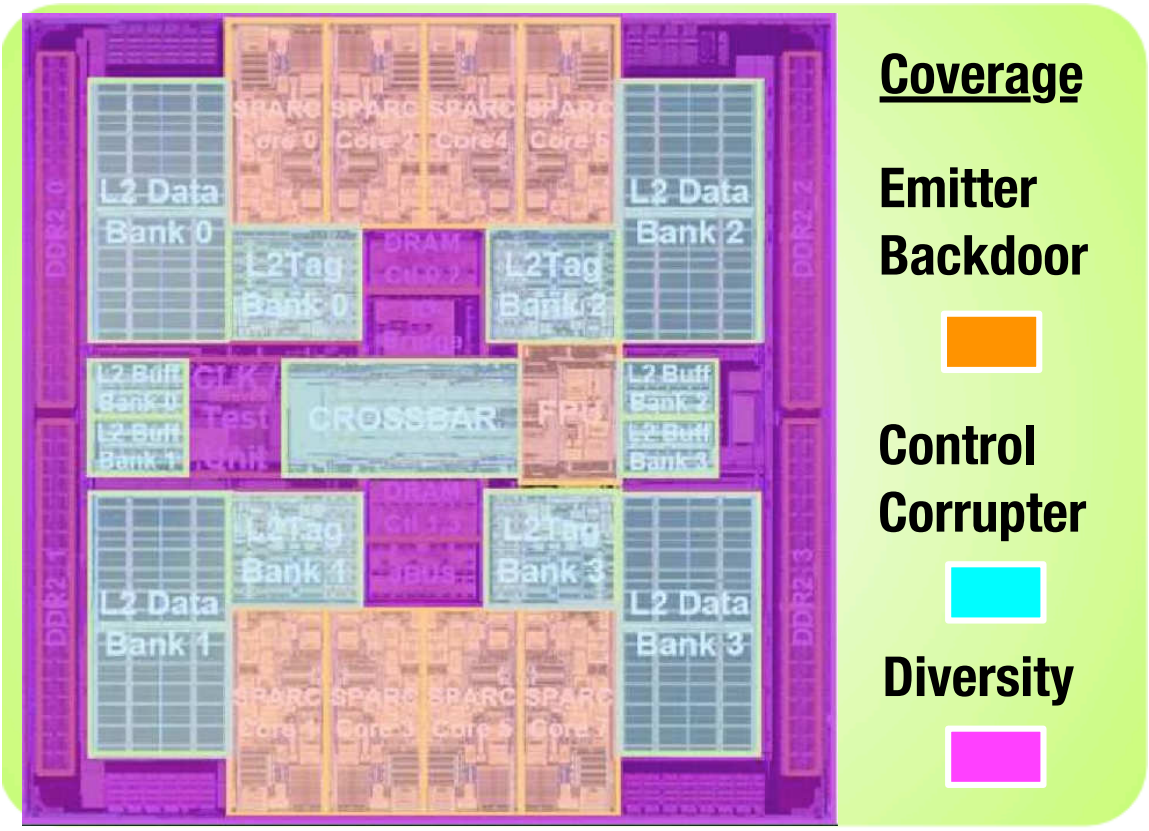

Figure 10.2: Illustration of units covered by **TrustNet** and **DataWatch** an OpenSPARC microprocessor.

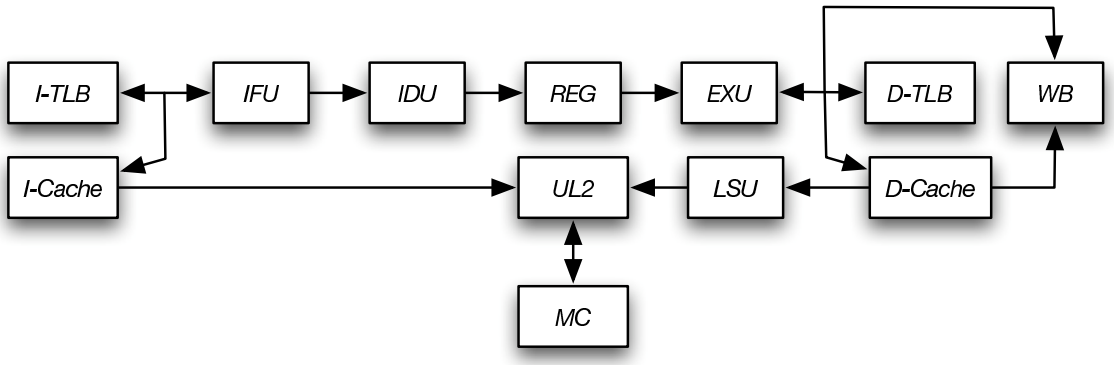

Figure 10.3: Units and communication in the hypothetical inorder processor used in this study.

In the case of branch and jump instructions, which do not go all the way through the pipeline, the information travels far enough for the EXU to know that a branch or jump is occurring. This monitor can be extended to support a speculative microprocessor if the monitor can reliably identify speculative instructions.

• *#2 IFU:* The primary responsibility of the IFU is to fetch instructions. Predicted by the I-Cache and reacted to by the IDU, this monitor confirms each cycle that a valid instruction comes out of the IFU if and only if an instruction was fetched from the I-Cache. This invariant catches any attack wherein the IFU sneaks instructions into the stream that did not come from the I-Cache. The monitor operates on the level of single instructions as opposed to whole cache lines. While the whole line is loaded into the I-Cache from the L2, the I-Cache knows when individual instructions are being fetched into the IFU.

• *#3 LSU:* The load-store unit (LSU) handles memory references between the SPARC core, the L1 data cache and the L2 cache. Predicted by the IDU and reacted to by the D-Cache, this monitor confirms each cycle that a memory action (load or store) is requested if and only if a memory instruction was fed into the LSU. This catches shadow load or shadow store attacks in the LSU. Our microprocessor uses write merging, which could have been a problem, since several incoming write requests are merged into a single outgoing write request. However, there is still a signal each cycle stating whether or not a load/store is being initiated, so even if several writes are merged over several cycles, there is still a signal each cycle for the monitoring system.

• *#4 I-Cache:* Predicted by the IFU and reacted to by the unified L2 Cache, this confirms each cycle that an L2 instruction load request is received in the L2 Cache if and only if that load corresponds to a fetch that missed in the I-Cache. The IFU can predict this because it receives an 'invalid' signal from the I-Cache on a miss. An I-Cache miss immediately triggers an L2 request and stalls the IFU, so there is no issue with cache line size. The IFU buffers this prediction until the reaction is received from the L2 Cache. This catches shadow instruction loads in the I-Cache.

• *#5 D-Cache:* Predicted by the LSU and reacted to by the L2 Cache, this is the same as the monitor #4 but watches data requests instead of instruction requests.

• *#6 L2 Cache:* Predicted by the I-Cache and reacted to by MMU, this is the same as

monitor #4 but is one level higher in the cache hierarchy.

• *#7 L2 Cache:* Predicted by the D-Cache and reacted to by the MMU, this is the same as monitor #5 but is one level higher in the cache hierarchy.

• *#8 D-Cache:* Predicted by the LSU and reacted to by the L2 Cache, this is the same as monitor #5 but watches writes instead of reads. It is necessary that two separate monitors watch reads and writes; if a single monitor counted only the total number of reads and writes, then an attacker could convert a write into a read unnoticed. This would cause old data to be loaded into the cache and prevent the new value from being written.

• *#9 L2 Cache:* Predicted by the D-Cache and I-Cache and reacted to by the MMU, this confirms that line accesses in the MMU correspond to line accesses issued by the level 1 caches. This monitor prevents shadow loads/stores executed by the L2 Cache.

Next we list the **DataWatch** monitoring triangles, also categorized by the untrusted unit being monitored:

• *#10 IFU:* Predicted by the IDU and reacted to by the I-Cache, this confirms each cycle that if the I-Cache receives a valid PC value it is the same as the value computed in the IFU. This required some duplication of PC logic but did not require any extra storage. This prevents attacks wherein the IFU maliciously changes the control flow.

• *#11 Data TLB:* Predicted by the checker data TLB and reacted to by the LSU, this confirms each cycle that the outputs of the data TLB match the outputs of the checker data TLB. This prevents data TLB attacks, such as permissions violations or page mistranslation. This is on the borderline of what we would start to call 'smart duplication' because the 'signatures' are so large. However, we included this is our **DataWatch** implementation and simulation.

• *#12 Instruction TLB:* Predicted by the checker instruction TLB and reacted to by the IFU, this is the same as monitor #11 but for the instruction TLB rather than the data TLB.

• *#13 IDU:* Predicted by the IFU and reacted to by the LSU, this confirms that the number of instructions decoded into memory operations matches the number of memory instructions fetched. For our microprocessor, this required that the IFU looked at a few bits

of the instruction. The monitoring occurs at a one cycle lag, so the timing on the critical path is unaffected. The IFU stores a few of the bits from the fetched instruction in flip-flops until the next cycle, when a prediction can be made with a few logical gates. For our case study, this is the only type of control corrupter decoder attack we address. The reason for this is that in our simple microprocessor, the only types of signals the decoder can cause are loads in stores (if, for example, the decoder changed an add to a subtract, this would be a data corrupter, because it would not alter the number of transactions in the execution unit, just the value of the output). In more complex microprocessors, decode units may be responsible for more types of transactions and might require additional monitoring triangles. When customizing a **DataWatch** system to fit a particular design, it is important up front to identify what types of signals each unit is responsible for.

10.8 Microarchitectural Details and Optimizations for Microprocessor Core Applications

While **TrustNet** and **DataWatch** can be adapted for a wide variety of hardware, we consider microprocessors to be a likely application. As a result, we develop some optimizations for keeping overheads negligible when applying **TrustNet** and **DataWatch** to microprocessor cores. The microarchitecture of our implementations for the predictor and monitor units are depicted in Figure 10.4. The predictor unit consists of (i) event buffers for delaying the issue of tokens to the monitor and (ii) token issue logic to determine when buffered events can be released from the event buffers. The predictor unit requires a small buffer because it is possible for multiple predictions to happen before a reaction happens, and these predictions must be remembered for that duration. These buffers can be sized *a priori* to avoid overflows. The monitor itself simply checks if events appear on the predictor and reactor inputs during the same cycle.

When designing the **TrustNet** system to catch emitter backdoors, we considered it to be important that the monitors fit directly into the pipeline without any complex timing or buffering issues. Since predictions and reactions must arrive at the monitor during the same cycle, timing must be controlled in the face of non-determinism, which arises in all

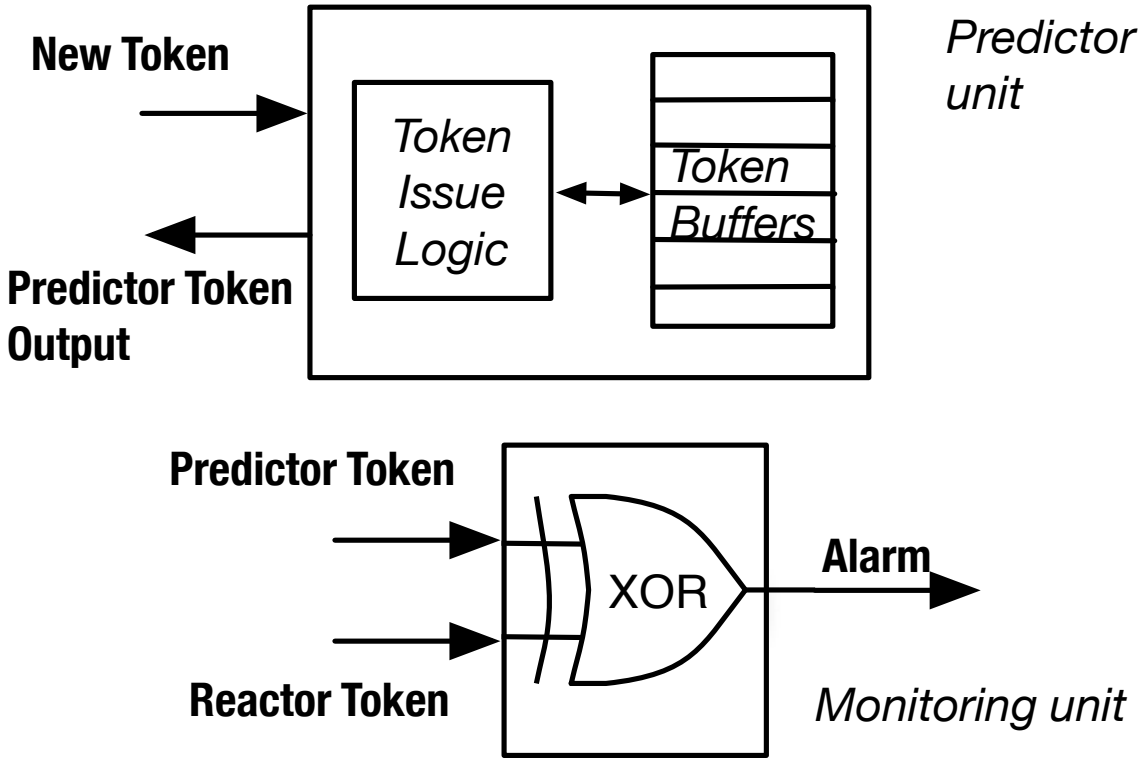

Figure 10.4: **TrustNet** Monitor Microachitecture.

microprocessors due to cache misses, etc. We handled this differently in the case of the memory hierarchy and in the case of the pipeline. The pipeline offers a natural lock-step manner for coordinating events. If a reaction stage is N pipeline steps down from a prediction stage, then the prediction stage has a size N buffer that advances only when that stage of the pipeline advances. Since the monitoring network advances in lock-step with pipelined events, timing is not a problem. For example, if the third pipeline stage wants to send a prediction to a monitor that lies in the fifth pipeline stage, this will take two pipeline advancements (no need for forwarding). If the third stage stalls for any reason, the prediction also stalls and gets buffered. When the data from the third stage reaches the fifth stage, the prediction token will also arrive. Of course, the prediction token should not pass through the fourth stage but should instead remain in the prediction buffer, with a bit denoting that it is semantically in the fourth stage.

In the case of the cache hierarchy, on the other hand, it is necessary to know which

predictions correspond to which reactions, because it is possible for memory requests to be handled out of order. This requires time-stamping of packets, for example with a one byte local time signature copied from an 8-bit modular counter.

Buffering in the Predictor Units Since predictors see events before reactors, a predictor may at times have made several predictions that have not yet been reacted to. For any monitor on any particular processor, there will be a known, finite bound on how many predictions can accumulate before a reaction occurs. We handled this by placing buffers on the predictors with enough slots to hold all the pending events in the worst case scenario.

A naíve solution for catching control corrupter backdoors in TLBs (translation lookaside buffers) is to simply have multiple designers build different implementations of the same TLB and compare their outputs each cycle (*i.e.,* DMR). Since TLBs tend to be power-hungry, highly or fully associative structures, duplication is an expensive approach. Instead of complete duplication, we propose a new TLB microarchitecture that provides significant protection without the costs associated with duplication. The TLBs contain page translation and permissions information not available elsewhere on chip. A TLB consists of a CAM that translates a virtual page into a physical page, which is then stored in a table (RAM) with the corresponding permissions information for that physical page.

Our idea is to create a direct-mapped 'checker' structure that has the same functionality as a TLB, the motivation being that a direct-mapped structure uses a small fraction of the power of an associative one. The TLBs in our case study are fully associative. We added functionality to the CAMs to output the line number of the output. This allowed us to build a checker TLB that uses these line numbers. Essentially, instead of having one CAM and a direct-mapped RAM (as is normal), we have one CAM and two direct-mapped RAMs that operate in parallel. The CAM provides matching entries to both RAMs in parallel. One of those RAMs communicates with the rest of the chip while the other RAM only gives outputs to a monitor (equality verifier). The equality check occurs at a one cycle latency, so the values are buffered for that cycle.

Naturally, the CAM could be tampered with so that it sends incorrect line numbers to the checker TLB. This would cause the equality check to fail because data from one line of the original TLB's RAM will be compared to data from a different line of the second RAM,

causing an alarm to be thrown. Therefore, our checker TLB turns a potential confidentiality or integrity attack into at worst an availability attack. We note that this availability attack would also be easy to catch at verification time because the passing of the line number is simple, combinatorial logic that can be checked by exhaustive enumeration.

While this duplication is more expensive than the simple **TrustNet** and **DataWatch** monitors used for backdoor protection, it is significantly less expensive than complete duplication and offers strong protection for a vulnerable structure.

10.9 Conclusions and Future Directions for Payload Detection Methods

Payload detection is an approach that we believe will always be necessary and will never be perfect. The necessity comes from the fact that we can never have 100% certainty in the perfection of our defenses. While proactive defense systems are desirable, reactive defense systems are necessary for the cases when proactive defenses are circumvented. The imperfection comes from the fact that as a reactive system, payload detection can only stop attacks that are more or less understand. If a brand new attack is developed that was never previously envisioned, it is unlikely to have a payload that has been accounted for.

We expect that payload detection will be valuable in practice as an efficient way to defend against classes of known or concerning attacks. For example, when developing a cryptoaccelerator, the set of known attacks is large and well understood. A few specific properties, such as making sure keys do not leak and extra ciphertext is not transmitted can prevent a large variety of likely attacks. Similar cases exist for other well-understood hardware, such as memory subsystems.

The limitation of payload detection that is essentially unavoidable occurs for modules that have not been well studied. For instance, it would be difficult to protect a brand new unit (or a piece of third-party IP) for which attacks have not been analyzed or proposed. One way to deal with this in the future could be to have a library of known interface compromises (*i.e.,* types of attacks) so that new hardware modules can be identified as fitting into one or more of these library classes.

Additionally, with dark silicon becoming increasingly prevalent, many optimizations of **TrustNet** and **DataWatch** (as well as duplication techniques are possible). In a setting where area is plentiful and power is scarce, a large amount of duplicate modules and/or monitors could be constructed and powered on only when necessary or psuedo-randomly.

Part IV

Fabricating Trustworthy Hardware

Chapter 11

Fabricating with End-to-End Security in Mind

An important point that we make throughout this work is that security approaches have to be based on globally-aware analysis of threats and personnel. One problem with the current state-of-the-art in security is that the hardware security space has been fragmented unnaturally into design-side approaches and foundry-side approaches. These two halves can both be more effective when combined together, but thus far they have been disparate and have used incompatible threat models.

In this part, we primarily seek to combine works, ideas and concepts from foundry-side security, along with our own novel contributions, to generate a coherent methodology for creating trustworthy hardware in the presence of both design-side attackers and foundry-side attackers.

We briefly argue for the impossibility of perfect design-side security without foundry-side security and *vice versa*.

Consider a perfect design that is not only backdoor-free but also contains unfailing runtime mechanisms for detecting any and all possible malicious behavior. This design is a document (a netlist) that is sent off to a malicious foundry. The foundry ignores the netlist and instead returns a chip produced from an older, less secure netlist. Thus, all the design-side security was for naught, as it never made it into the physical chip.

Consider a perfect foundry that not only avoids backdoor insertion but also has perfect chip imaging to ensure that all fabricated chips precisely match the provided netlist. A malicious design house gets malicious IP included in the netlist. The foundry perfectly and precisely fabricates a chip from the malicious design, thus producing the backdoor inserted by the malicious design house. Thus, the foundry-side security was for naught, as it only served to produce all malicious circuits hidden in the netlist.

Therefore, we consider it necessary to be aware of both design-side and foundry-side security when developing hardware. In the following chapters, we discuss how foundry-aware techniques can work synergistically with the design-side techniques discussed thus far and be used to develop trustworthy hardware in the presence of a malicious conspiracy that includes both malicious designers and a malicious foundry.

Chapter 12

Beacons: A Novel Power-Based Attestation Mechanisms

To achieve our goals, we first propose a new attestation mechanism, which we call a *beacon*, that allows for the usage of side-channel measurements post-fabrication that attest to design-side properties.

12.1 Enhancing Design-Level Protections with Beacons

Given that there exist runtime systems for dynamically protecting against design-level backdoors, we want our foundry-level defenses to work together with those systems. In our methodology, these design-level protections are combined (post-design phase, pre-fabrication phase) with the beacon that prevents attacks from a malicious foundry. The process begins in the design phase with anti-backdoor circuits and ends post-fabrication with the attestation of the beacon. We next explain the details of this system.

Our power attestation construct – a beacon – is a digitally controllable challenge/response circuit. The output is unpredictable to anyone except the owner of the key used to generate the beacon. The response of the beacon can be either digital or analog, and we consider both options, though we ultimately use analog (specifically power). The idea of beacons is illustrated in Figure 12.1.

An analog side-channel beacon is a substantial and predictable side-channel emanation

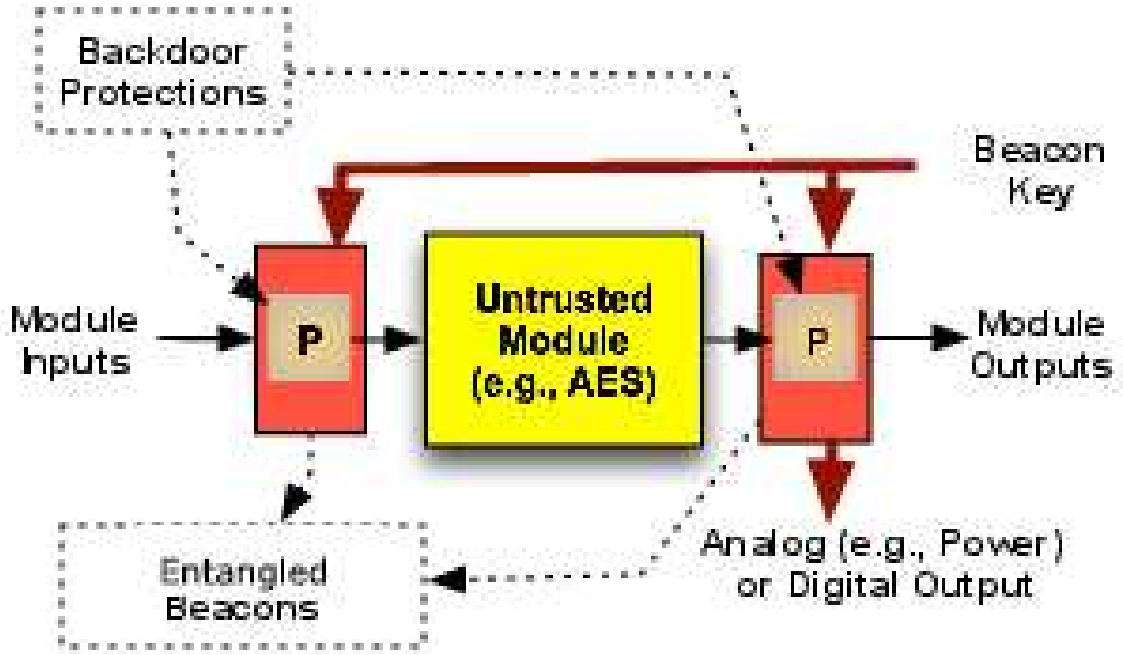

Figure 12.1: An overview of how a beacon works. Beacons are added to the backdoor protection circuits and entangled with the protections. When a beacon key is supplied, it outputs an analog power signature or a digital output that can be used by the auditor.

caused by a digital input. It can be thought of as a spot on the chip that can be intentionally made hot under controlled circumstances. The side-channel emanation should be easy to measure and reliable in its values. We achieve this by using power as our side-channel throughout this chapter. We consider the following requirements to be necessary for the practical application of power side-channel beacons:

- Average power usage of the design should not be impacted by the inclusion of beacons.
- Peak power constraints of the design should not be violated by beacons.
- The difference between power usage when a beacon is on versus when it is off should be reliably measurable and thus should be large with respect to the total power of the system: too large to be by accident (e.g., process variations) or to go undetected.
- The trigger for the beacons should be an unlikely input to prevent the foundry from guessing the trigger to learn and replay the expected power output.

Our conclusion from these requirements is that the natural solution for a beacon is a

digitally-triggered circuit that raises power by a substantial amount that is at most the difference between average and peak power. The triggered execution aspect of the beacon is functionally similar to a typical backdoor even though it is used benignly for the purposes of security. We delve into the implementation later in this chapter, but we give now the intuition for why all the requirements are met by this solution.

• Average power is unaffected because the beacon, being a key-enabled circuit, only needs to be on for a controlled and small number of cycles. During normal operation it is off.

• The beacon raises power usage from average power to a controlled level above average that is at most peak power.

• The difference in power is on the order of the difference between peak and average, which in any realistic system is large and readily measurable. This difference can be chosen to be smaller if desirable as long as it is big enough to be observable.

• The trigger for the beacon (just like a trigger for a backdoor) can evade detection, this being one of the fundamental motivations for this field of study in the first place. The beacon is applied after backdoor defensive mechanisms are in place and is thus unaffected by the defensive mechanisms themselves. A beacon can be thought of as a beneficial backdoor applied by the last actor in the design process.

A *digital beacon* is the same concept as the analog one but with digital outputs instead of side-channel emanations. It is an intentional backdoor that outputs a signal (*e.g.,* a one) when it sees a special key and a different signal (*e.g.,* a zero) for all other possible values. There are nearly unlimited ways in which a digital beacon output can be relayed to the auditor. It could be placed into a pre-chosen register, put in memory, sent out directly through a pin, or made externally visible through any other number of means.

The trade-off between the digital approach and the analog side-channel approach exists in detectability for both the attacker (malicious foundry) and the auditor. The digital approach is more easy to detect, thus making it easier to implement and easier for an attacker to circumvent. Using a digital output might offer the attacker the possibility to backtrace from the output pin to identify some of the logic [Helfmeier *et al.*, 2013]. Such a method would not be possible in the case of an analog side-channel beacon, because the side-channel beacon does not connect to any output pins. From here on out, we go with

the power side-channel approach.

12.2 Relationship Between Beacons and IC Fingerprinting

Integrated Circuit (IC) fingerprinting is an approach to a related problem. Work in the area attempts to find emergent power signatures that convey the identity of a design and then detect anything that is different (*e.g.,* a possible backdoor) [**?**]. Additional work has been done toward enhancing the ability to detect power signatures using dynamic monitoring [Narasimhan *et al.*, 2012].

IC fingerprinting is difficult in practice because foundry-inserted backdoors can be very small and might have almost no impact on power. For this reason, false negatives and false positives are a major area of study for IC fingerprinting.

Beacons, on the other hand, are the opposite of IC fingerprinting in some respects. Instead of taking a design as given and attempting to detect small fluctuations in power caused by malicious circuits, we instead intentionally entangle the beacon in the design and ignore small fluctuations. At a high level, fingerprinting attempts to prove that two things are the same, whereas beacons attempt to show that two things are different. Proving sameness turns out to be very hard, as demonstrated by prior work, because differences can be almost arbitrarily small and noisy. By switching from a hard problem to a relatively easy problem, we are able to do away with false positives and false negatives and avoid the need for noise filtering.

12.3 Beacon Construction and Implementation

The number of ways in which a beacon can be constructed is almost unbounded. By varying the choice of logic gates (AND, OR, etc.) the functionality required of the beacon can be provided in exponentially many different ways. The template we choose for a side-channel beacon is depicted in Figure 12.2. There are two core components.

• A state machine controller. The state machine simply detects when the auditing key has arrived as an input. In the simplest case, the logic is essentially a comparator that has been

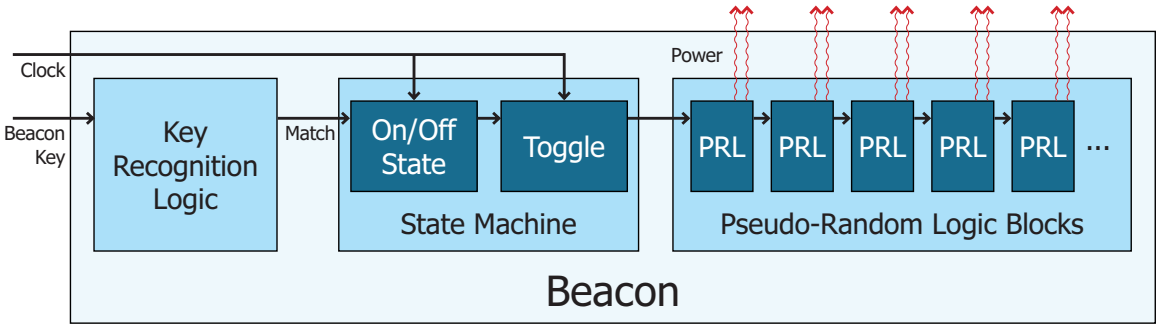

Figure 12.2: A beacon implementation. The state machine control is primarily a comparator that looks for the secret key. When it does, the state switches from zero to one, turning on the rest of the circuitry. The toggle bit changes every cycle, causing the combinational logic to flip every cycle. Within each pseudo-random logic block (PRL), there is randomly generated logic. When the state is zero, the activity factor is zero. When the state is one, the activity factor is one.

entangled (using an algorithm to be described later).

• Pseudo-random logic blocks. The other core component is a set of pseudo-random logic blocks (PRLs). A PRL is a circuit of depth one: it is an array of logic gates that are fed inputs and give outputs in parallel. To construct the beacon power circuit we tile multiple PRLs next to each other to form a grid of logic blocks. Each PRL passes along the input signal through randomly chosen logic blocks. Each pair of adjacent PRLs forms a bipartite graph, where each edge either exists or does not exist based on a pseudo-random decision based on the secret key (the connections are formed during hardware synthesis).

How do these PRLs work? We will illustrate this by assuming that the each PRL is a long column of NOT-equivalents.[1] We make this choice to make it easy to control the activity factor, but other choices would be acceptable as well. The circuit begins operation by broadcasting a bit (or bits) to the first PRL.

The visual to have at this point is that the PRLs are like a long chain of see-saws. The reason the visual is a see-saw is because the signal flips when it moves from one PRL to the

[1]By NOT-equivalents, we mean gates that do the same thing as a NOT gate. This could include multiple implementations of NOT gates, NAND gates, NOR gates, and a variety of other choices.

next (because they are all NOT-equivalents in the simplest case). If the signal is all ones in the first PRL, it is all zeros in the second PRL, all ones in the third PRL, and so on. When the toggle bit flips, all of the one-PRLs go to zero and the zero-PRLs go to one, causing 100% of the wires to toggle.

If we wanted to create a hot spot, we could have used practically any circuit, but why PRLs? In our experience, the activity factor in an average circuit normally ranges between around 0.01 to 0.1, but with PRLs we can use a circuit that is 10 to 100 times smaller for the same power surge by guaranteeing that the activity factor is at or near one (100% activity).

This type of beacon has fairly clear power costs. There is the leakage power (power consumed when the circuit is in standby) and the dynamic power (power consumed when the circuit is operational). The leakage power comes from the two flip-flops and the PRLs. If a flip-flop has leakage F and a PRL has leakage P, then the leakage is roughly

$$2F + NP$$

where N is the number of PRLs used. If each PRL has a maximum dynamic power consumption of D, then the power consumption while the beacon is active is roughly

$$2F + N(P + \alpha D)$$

where α is the activation factor. Since we want the beacon consumption to be large compared to the leakage power, we want the ratio

$$\frac{2F + N(P + \alpha D)}{2F + N(P)}$$

to be large. The two ways to make them happen are to make α and D large. While there are multiple ways to achieve this goal, we presented one set of options. This set contains an exponentially large[2] number of possible beacon circuits, each with maximal activity factor ($\alpha = 1$).

[2]The number of distinct beacons within the set is exponential in the size of the random key.

As we will see in Section 9.3.2, cost and power can be estimated analytically and evaluated empirically for different beacon sizes. By increasing the size of the PRL grid, the intensity of the hot-spot is increased.

Chapter 13

Entanglement-Based Methods for Preventing Counterfeiting and Reverse Engineering

Given a beacon that can attest to security properties of a fabricated hardware device, our goal is to prevent an intelligent attacker from undermining the beacon. Without learning the key, it is impossible to fake the power signature; thus the only attack vector for an adversary is to uncover the beacon key and create a fake beacon before or during fabrication. For this reason, recovering the key becomes an exercise in reverse engineering, because with a full understanding of the circuit at the most detailed level, an adversary can potentially guess the key.

Our technique for embedding beacons and protecting against key extraction through reverse engineering is depicted in Algorithm 5. Our system has three pieces: a beacon, a protection mechanism and an untrusted module. At a high level, the beacon and protection are entangled together, while the untrusted module can operate correctly only when a special key (K_M) is supplied to it. This module key, K_M, is generated only when both the beacon and protection have not been tampered with in any way; entangling the beacon circuit and the protection circuit ensures that they cannot be separated by an adversary to leave the beacon intact without the protections. Next, we describe individual portions of our design

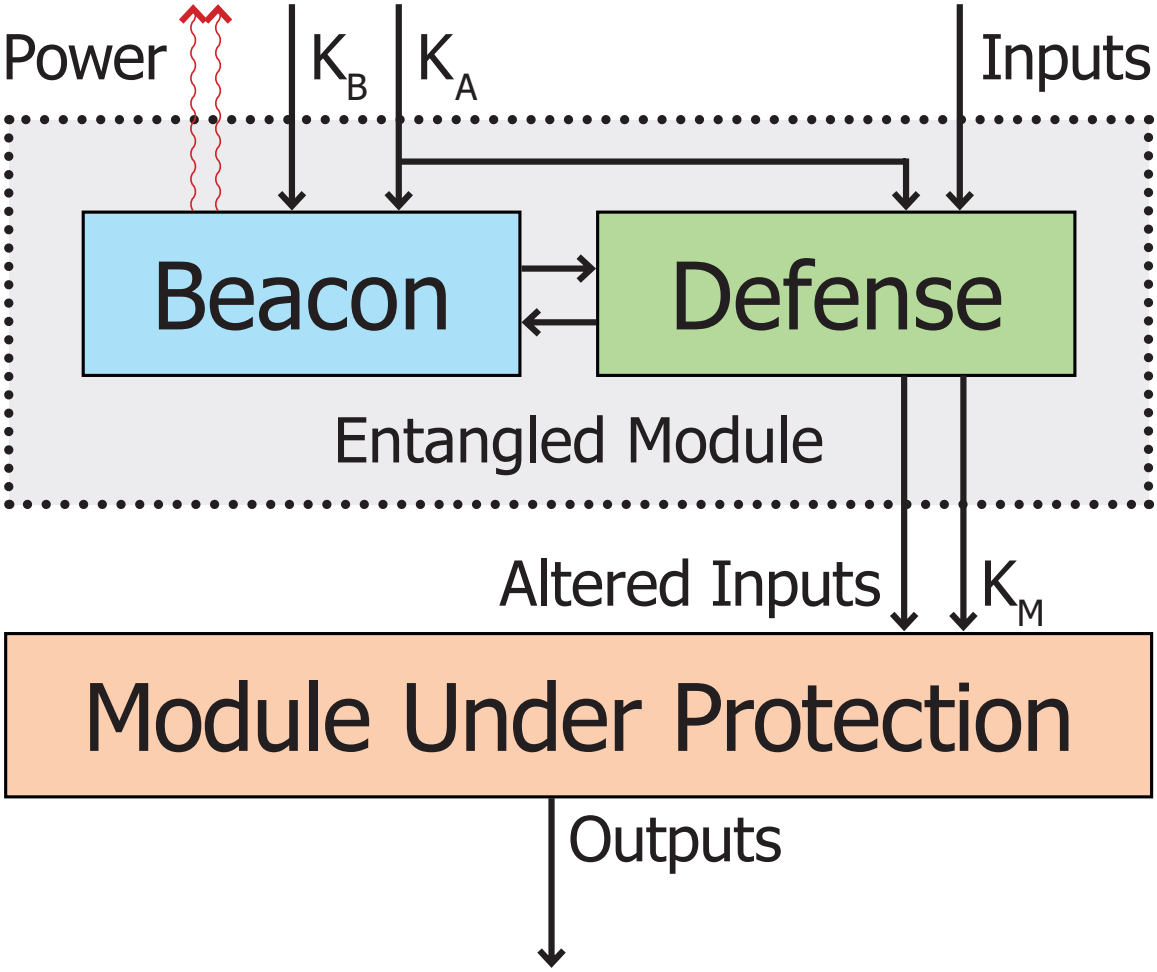

Figure 13.1: An overview of the incorporation of an untrusted module, using an entangled protection and side-channel beacon.

viz. our techniques for entanglement and module key generation.

13.1 Key-Based Activation

How can we entangle two circuits – the beacon and the protection circuit – in a way that is difficult for an adversary to separate them out even when he has complete access to the netlist? Simple mixing is dangerous: by construction, the beacon circuits will be relatively dark for all but one of an exponential number of possible inputs, while the protection circuit will ordinarily show some activity for all inputs. An intelligent adversary can take advantage

Algorithm 5 Combine beacons with protections and modules

```
1: for all security-critical modules m do
2:     K_A ← Activation Key for m (public after fabrication)
3:     K_B ← Beacon Key (private)
4:     K_M ← Module Key (private)
5:     B ← newBeacon(K_A, K_B)
6:     Synthesize B
7:     P ← protection(K_A)
8:     Synthesize P ∪ B
9:     for all bits w in K_M do
10:        w ← randSelect(value(w), P, B)
11:    end for
12:    Synthesize P ∪ B ∪ M
13: end for
```

of this difference to identify and snip out the protection circuit, leaving the beacon intact[1].

One solution to this problem is to match the activity of the beacon with the activity of the protection circuit. In practice, this solution is hard to implement because the activity of the protection circuit depends on its inputs. To obtain the same activity for all possible inputs, the protection and beacon circuits must be functionally identical which is not often the case. Of course, one can try to match the activities for many cases but it can be argued that any small difference in the unmatched cases can be exploited by a sufficiently capable adversary to differentiate the two circuits.

Instead of trying to match the activity factors or trying to make them look similar, our solution is to take away the attacker's ability to turn on the circuits before fabrication. What we propose is to prepare the circuits so that they can be turned on only when another special key – called the activation key (K_A) – is supplied to the beacon and the protections. This key is kept secret from the foundry during manufacturing and can be provided through a single interface by using the same interface for the entangled beacon and protections. This

[1]We do not know of such an attack for advanced technology nodes, but we expect this to be possible with some degree of difficulty for large circuits.

is important because it means an attacker cannot simply trace the input pins to learn about the design protections and how they are entangled with the beacons.

This method for turning on circuits is known as key-based activation technology and has been proposed previously to protect against IP piracy [Roy *et al.*, 2008; Rajendran *et al.*, 2012]. The idea is best explained with a toy example: imagine that every intermediate wire in a design is XOR'd with a special input bit. If that bit is a zero, the design works. If that bit is a one, the design essentially produces random noise. In this toy case, only by supplying all zeros can the design be made to function correctly. By introducing XOR and XNOR gates (or arbitrary functional equivalents) into designs, any design can be converted to a design that works only when some special inputs (an activation key) are supplied.

Our approach for beacon construction is shown in Algorithm 6. Note that K_B is the key to light up the PRLs to maximum activity.

Algorithm 6 Entangle a Beacon

1: $K_A \leftarrow$ Activation Key (public after fabrication)
2: $K_B \leftarrow$ Beacon Key (private)
3: $B \leftarrow$ state machine to recognize K_B
4: **for all** bit w in K_A **do**
5: $\quad r \leftarrow$ random internal wire in B
6: $\quad$ Insert XOR/XNOR equivalent to recognize value(w) at r
7: **end for**
8: Re-synthesize B with NAND logic

We make two additions to prior key-based activation technology [Roy *et al.*, 2008; Rajendran *et al.*, 2012]. We are concerned that an intelligent attacker could deduce (or guess) the value of an input bit (part of K_A) based on the logic the input bit feeds into. For example, in a toy case, XORs could represent 0-bits and XNORs could represent 1-bits, making the attacker's job far too easy if the attacker can identify the gate types by examining the netlist. Semantically, a 0-bit should feed into an XOR followed by some arbitrary logic, while a 1-bit should feed into an XNOR followed by some arbitrary logic. We first insert at least one negating gate (NOT, NOR, NAND, etc.) after each XOR so that every entry point is capable of processing either a 0 or 1 bit. Secondly, we resynthesize the whole

beacon and protection using one type of gate[2] to enhance uniformity and prevent these gate-identification attacks. We do not know if the aforementioned attack could be made to work against prior work.[3] However, with these minor additions, we protect ourselves against the attack in case it ever becomes feasible.

13.2 Module Key Generation

Once we have a working beacon, we synthesize it together with a protection mechanism (activated with the same key K_A). Next, we want to ensure that an adversary does not leave the protection circuit and beacon intact while somehow replacing the untrusted module with another module that defeats the protection module.

Our solution here is to use key-based activation again. However, instead of using K_A, we derive the activation key for the untrusted module from the circuits in the protection and beacon modules. This derivation ensures that the untrusted module will not work with a compromised protection or beacon module. The key derivation is simple: we use intermediate wires from the modified beacon and protection circuits to pull out the bits needed to produce a new derived key K_M. The wires are chosen such that when K_A is supplied, K_M is deterministically produced, while in all other cases the outputs are *don't-care bits* (values not equal to K_M).

13.3 Security Analysis of Netlist Entanglement

Difficult to Attack: The main guarantee of our system is that an attacker cannot efficiently reverse engineer the circuit under protection and thus cannot counterfeit it or include backdoors. While the entry points of all bits in the keys are known (*i.e.,* the locations of the gates are known), brute forces the actual values of the key bits (zeros vs. ones) is inefficient due to the exponential explosion of possibilities. Prior work [Rajendran *et al.*, 2012] has

[2]We arbitrarily chose NAND logic.

[3]For this reason we do not apply this extra step to the module under protection. However, if desirable, the same step could be applied.

shown that recovering keys can require as many as $10^{44} \approx 2^{146}$ test patterns, which is far too many for an attacker to perform, even with tremendous resources.

Composable: While our system is composed out of components that rely on entanglement keys, the composition does not weaken the system. In order to attack the system, an attacker must recover K_A or K_M or both. K_A is applied in the same way as prior work and thus has equal strength. K_M is derived from K_A using a pseudo-random one-time function. Thus, if K_A is not recovered, recovering K_M reduces to the same problem. Therefore, breaking our system is *at least* as hard as breaking the state-of-the-art. Thus, our system composes without weakening the state-of-the-art.

Lifetime of Keys: The lifetime that a key has to be kept secret is from when they are generated (post-design) to when attestation is performed (post-fabrication). Once attestation tests have passed, the keys can go public, and in practice they probably would so that they could be included in firmware or something similar. The reason keys can go public at that point is that once the hardware has been manufactured, it is too late for an attacker to go back and attack the netlist.

We briefly summarize the three keys used in our system for clarity.

• K_A, the activation key, is kept private until after manufacturing is complete, at which point it can be loaded into firmware or distributed with the product. Once the manufacturing is complete, K_A no longer has any need to be kept a secret. Without K_A, the behavior of the design is undefined.

• K_B is the beacon key. This is a private key, used only during auditing (post manufacturing) to turn on the beacon side-channel for attestation. Without K_B, the functionality of the beacon is undefined.

• K_M is a private key that enables the untrusted module. Without K_M, the behavior of the module is undefined. K_M is derived from K_A. Like K_A, K_M must be a secret prior to manufacture. After manufacture, it is acceptable for attackers to use chip-probing methods to uncover K_M, because it no longer has value at that point in time, as the device has already been manufactured.

We note additional security-related points:

• From the perspective of an attacker, reverse engineering a beacon is equivalent to uncov-

ering both K_A and K_B, which is equivalent to guessing a 128-bit key. In our design, neither key exists physically in memory on the chip; the keys are implicitly coded in the circuits.

• The key K_M used for the module under protection is equivalent to a standard entanglement key. Under the same axioms that protect K_A and K_B, K_M cannot be easily guessed. K_M can also be made larger (more bits) than K_A if desirable, as the mapping from K_A to K_M does not need to be surjective. This means that the foundry cannot easily apply parametric backdoors or other foundry-specific attacks to the module under protection. Parametric backdoors have become increasingly studied and can be difficult to detect if properly hidden [Becker *et al.*, 2013].

• Post-attestation, if a second fabrication run is needed, new keys should be chosen. If this is a financial concern, one possible approach to avoid re-synthesis is to compare the production from the second run (using the same mask and not changing keys) to the devices produced in the first run. Prior work has been done in this area toward making this comparison efficient and practical [**?**; Jin and Makris, 2008; **?**]. As an alternate possibility, there has been recent work on building inspection resistant memory [Valamehr *et al.*, 2012], though expanding that to full designs and making it robust would require further research.

Chapter 14

Evaluation and Analysis of Beacons

In this chapter, we analyze the costs and trade-offs for a power-based beacon. We also include a variety of practical case studies.

Our designs are all synthesized using a Synopsys tool chain and standard 90nm ASIC libraries. We use the Synopsys power analysis tools to acquire power information, in addition to area and timing reports. We alter the power magnitude by adjusting the size of the beacon. The important parameter is the size of what we call the *beacon grid*. This grid is the matrix formed by tiling the rows of PRLs in the beacon. A beacon made out of x rows of height y can be referred to as an x-by-y beacon. The leakage power drawn scales with both the two flip-flops and the logic in the beacon grid (see Figure **??**). Ignoring second-order effects, the leakage power is roughly

$$2L_f + L_c + xyL_g$$

where L_f is the power leaked by a flip-flop and L_c is the power leaked by the comparator. L_g is the power leaked by an average gate in the grid. All of these costs are linear, which is as good as we can hope to do. The additional dynamic power usage that we can achieve scales as roughly

$$\alpha L_c + \alpha xyL_g$$

where α is the activity factor (which is kept high by design in our implementation) of the

combinational logic. The activity factor for the logic governing the flip-flops is effectively zero.

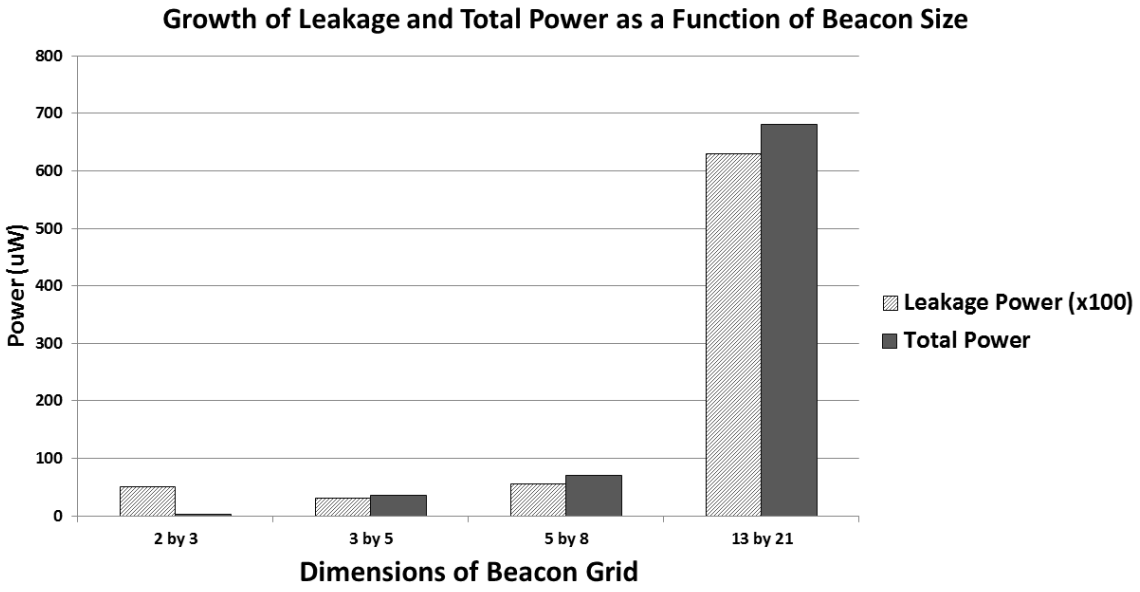

Figure 14.1: The leakage and total power draws achieved by different sizes of beacon grids. The first dimension refers to the number of rows and the second dimension refers to the height of each row.

In Figure 14.1, we show the leakage and total power for different sizes of beacons. These values can also be impacted by physical decisions about the gates, such as drive strength, as shown in Figure 14.2. The ratio of height to width for the beacon grid is an arbitrary choice. We went with the 'golden ratio' for lack of a better choice. As Figure 14.1 shows, the constant leakage cost of the flip-flops matters only for extremely small beacons. By increasing the size of the beacon grid, the dynamic power draw can be brought up rapidly. Note that the bars for leakage power are multiplied by 100 in size to make them visible. The leakage power drawn is fairly negligible.

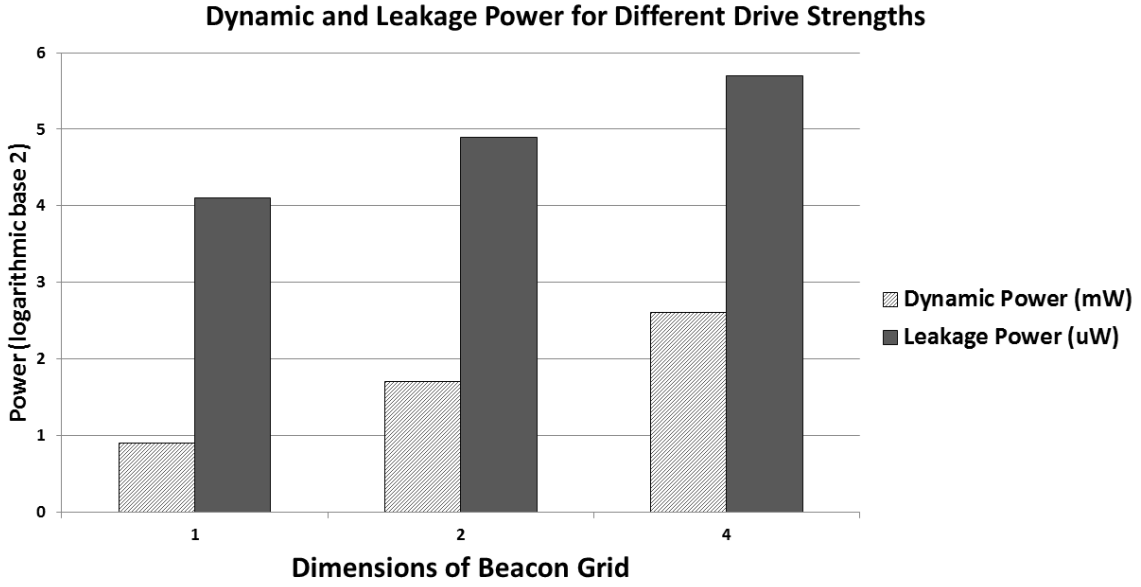

Figure 14.2: The leakage and dynamic power draws achieved by different drive strengths for a simple beacon. Higher drive strength can allow for more dynamic power from a fixed number of gates. Note that the leakage power is in microwatts, while the dynamic power is in milliwatts, so the leakage power is small.

14.1 Case Study: Applying Beacons to Payload Detection Systems

In order to test our methodology, we implement **TrustNet** and apply a beacon to it. In this case study, we show how beacons can complement and harden **TrustNet** (and similarly **DataWatch**). Without a beacon in place, **TrustNet** by itself would be relatively easy to disable via a malicious foundry. Assuming the gates — which are *a priori* known — could be located, the disabling could be done by severing a single wire or causing a single stuck-at-one fault. We examine three different applications of **TrustNet** to study costs of different deployment conditions. The three cores we examine are a cryptographic accelerator, a small general purpose core and a larger general purpose core.

For example, consider an AES encryption accelerator. One invariant is that the AES encryption unit should output the same number of bytes of ciphertext as it receives as plaintext. If the amount of ciphertext exceeds the amount of plaintext, that implies wrongdoing

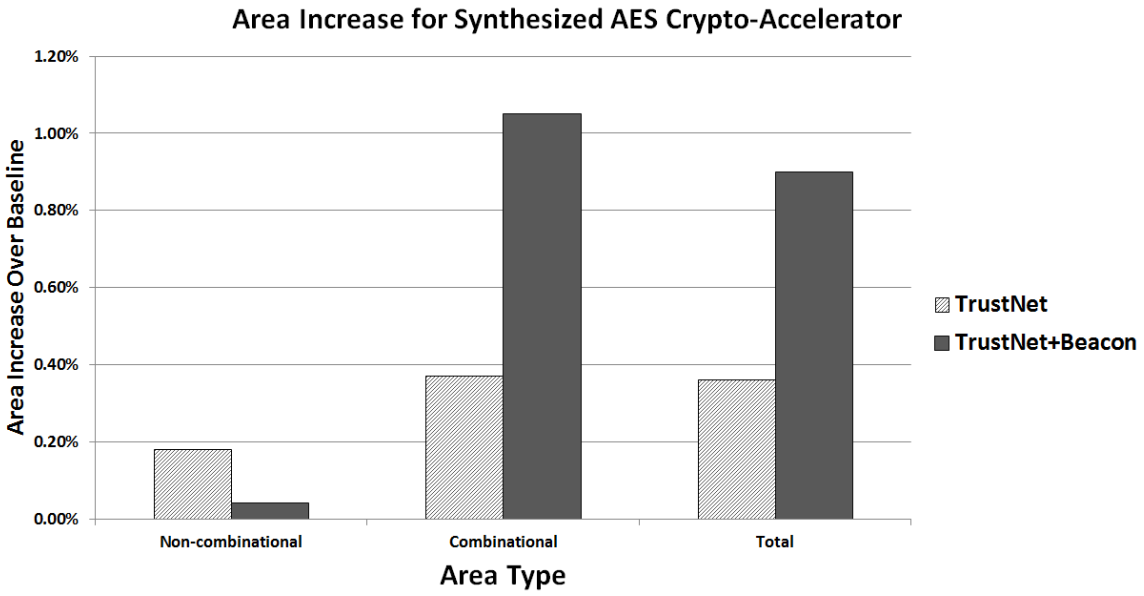

Figure 14.3: Area overheads of the **TrustNet** defense mechanism and beacon placed in a custom AES crypto-accelerator. Overheads are shown as a percentage of the baseline system. Timing requirements are maintained in all three designs.

Table 14.1: Specifications for the Two Chosen FabScalar Cores

Core Type	Out-of-Order	Fetch Width	Fetch Depth	Execution Units	Issue Queue Size
Small	Yes	1	1	4	32
Large	Yes	6	2	6	64

and could mean a key or some other private data is being leaked.

14.1.1 Design Case A1: An AES Accelerator

An AES accelerator is an interesting case for the necessity of hardware security. By definition, such accelerators are given privileged access to secret data. This makes them prime targets for attack by malicious entities. For our case study, we implement a simple and effective emitter backdoor into a custom AES accelerator design. We build the four components necessary to demonstrate a **TrustNet**-aware beacon:

- A custom AES accelerator, designed in the Verilog HDL.
- A hidden emitter backdoor in the AES design.
- A **TrustNet** module that disables emitter backdoors by checking that the in-packets match the out-packets cycle for cycle in the AES design.
- An entangled beacon that attests to the presence of the **TrustNet** module.

Figure 14.3 shows the overheads associated with **TrustNet** and with adding a beacon to **TrustNet**İn order to implement **TrustNet** to meet the timing requirements of our custom AES unit, we had to add a small amount of pipelining to the original design of **TrustNet**. We were able to make the beacon meet timing requirements in one case by adding a single pipeline and as an alternate case with clock division. For the results shown, we went with clock division. We find, as expected, that the non-combinational area overheads are nearly zero. The combinational area overhead of the beacon is comparable to that of **TrustNet** itself.

14.1.2 Design Case A2: Out-of-Order Processor Cores

To study out-of-order general purpose cores, we make use of FabScalar, a framework for automatically generating implementations of microprocessors [Choudhary *et al.*, 2011]. It is capable of generating cores with different sizes and features (such as different fetch widths, commit widths, number of execution units, issue policies etc.) The two cores we use for the bulk of our experiments are specified in Table 14.1. An architectural view of a FabScalar core is portrayed in Figure 14.7. The security in **TrustNet** is based on finding invariants of hardware designs. Many invariants of FabScalar cores are true for all generated cores, regardless of the given parameters. We add **TrustNet** to the FabScalar framework, allowing the automatic generation of out-or-order cores that have built-in security checks.

Figure 14.5 shows the area overheads associated with adding the **TrustNet** defense modules and also adding a beacon that attests to the presence of **TrustNet**. This data is for one of the smaller FabScalar cores. As we can see, since the beacon overhead is smaller even than that of the backdoor defense, it amounts to only a small fraction of one percent. We also note that the overhead of a beacon is amplified by the choice of activation method.

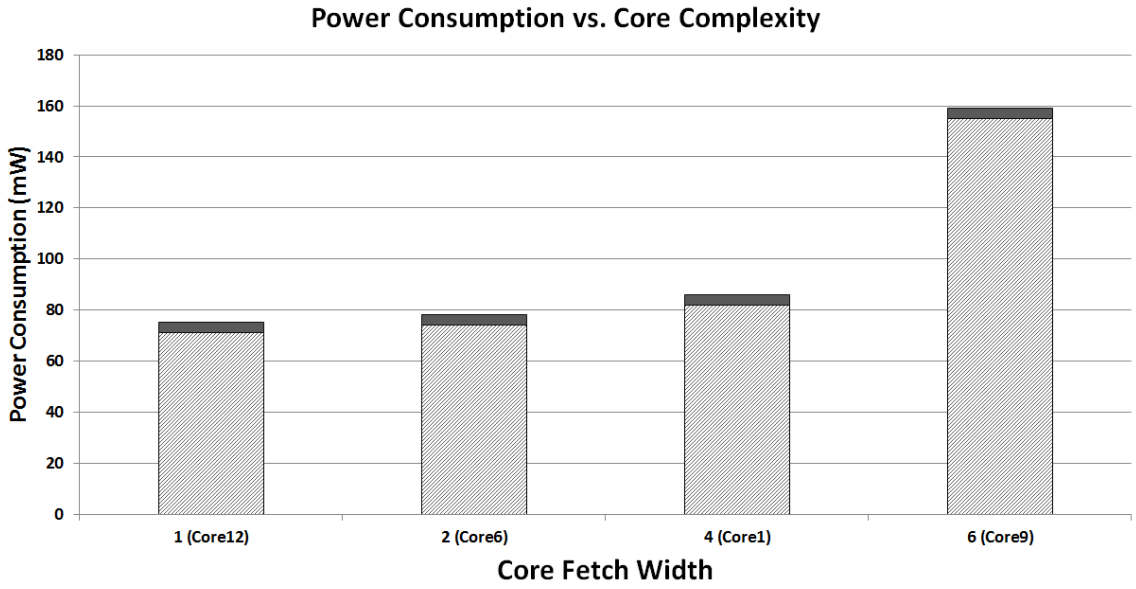

Figure 14.4: The trend of how the cost of a beacon scales from smaller to larger cores in the FabScalar family. The labels on the bars are the names these cores have within the FabScalar family. The numbers in the names given within FabScalar have no relation to the sizes of the cores. Naturally, if the beacon size is roughly constant, the overhead diminishes when compared against power consumed by larger and larger cores. In short, the closer a design is to the power wall, the cheaper it is to use a beacon.

Applying key-based activation increases the total overhead in this case to 1.7%. We also performed a study with a larger FabScalar core; the results are shown in Figure 14.6. Since this is a larger core, the impact of the beacon is much smaller as a percentage.[1] Finally, the costs — both area and power — diminish relatively as the chip scales to larger sizes (Figure 14.4 shows the power case).

[1] Automatic synthesis is a complex and largely automatic process; slight variations can result in small changes in area. This is the reason we sometimes see slightly negative area overheads.

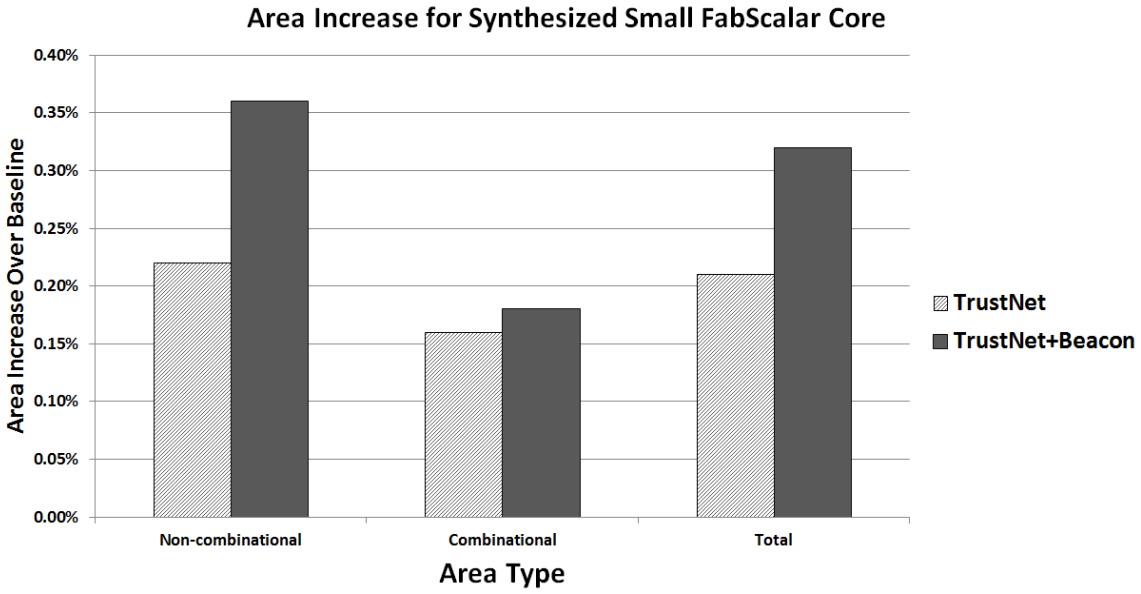

Figure 14.5: The area overheads attributed to the **TrustNet** defense system and a corresponding beacon for the smaller FabScalar core.

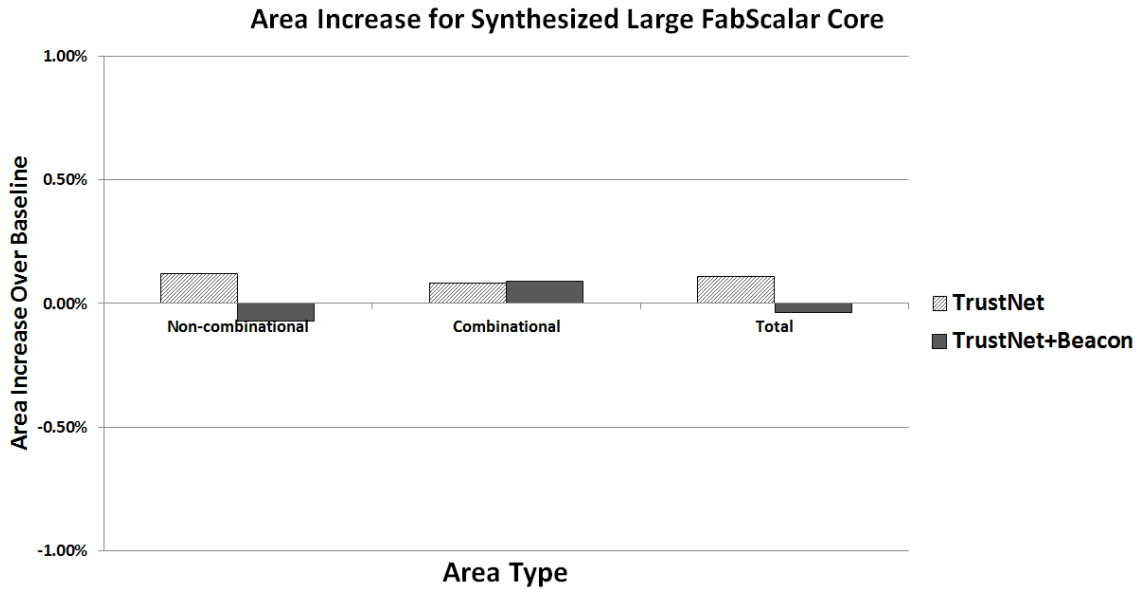

Figure 14.6: The area overheads attributed to the **TrustNet** defense system and a corresponding beacon for the larger FabScalar core. Slight negative values are to be expected due to the chaotic nature of the synthesis algorithms.

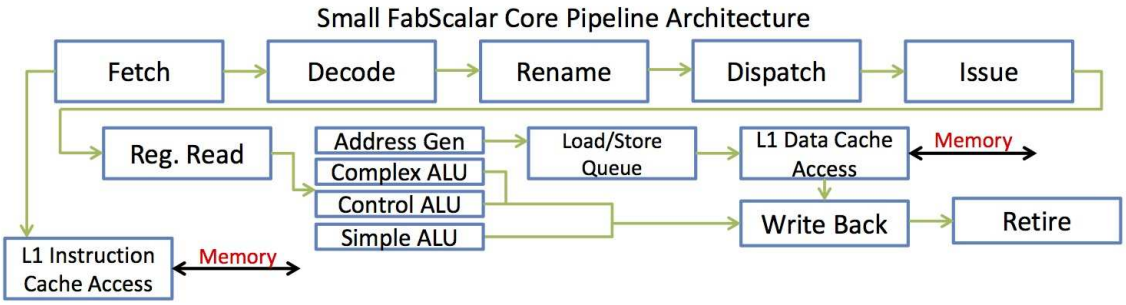

Figure 14.7: A high level view of the FabScalar auto-generated processor architecture.

14.2 Case Study B: Applying Beacons to Trigger Obfuscation

As a second case study of the beacon methodology, we consider combining beacons with trigger obfuscation. This is another example — like **TrustNet** — of a defense mechanism that consists of a small piece of circuitry that could potentially be identified by a malicious foundry. While it is more complex than **TrustNet**, re-routing a few key wires in the defense logic could disable the effects of the defensive system. We consider the impact of beacons as a complementary tool to trigger obfuscation, using a custom AES crypto-accelerator as the baseline design.

As shown in Figure 14.8, the area impact of the beacon is negligible. Trigger obfuscation is a more heavy-weight defensive mechanism, and the impact of the beacon on area falls into the noise. All timing constraints are still met by the design.

14.3 Limitations of Beacons

The notion of entangling or encrypting computational behavior (including hardware, software and Turing Machines) has been discussed in theory literature. There have been important negative results that highlight the limitations. These results have demonstrated that some limitations on entanglement are unavoidable.

We note that entangling two circuits is a form of obfuscation. In [Barak *et al.*, 2001], it was proven that the most widely accepted, generic notion of obfuscation is impossible

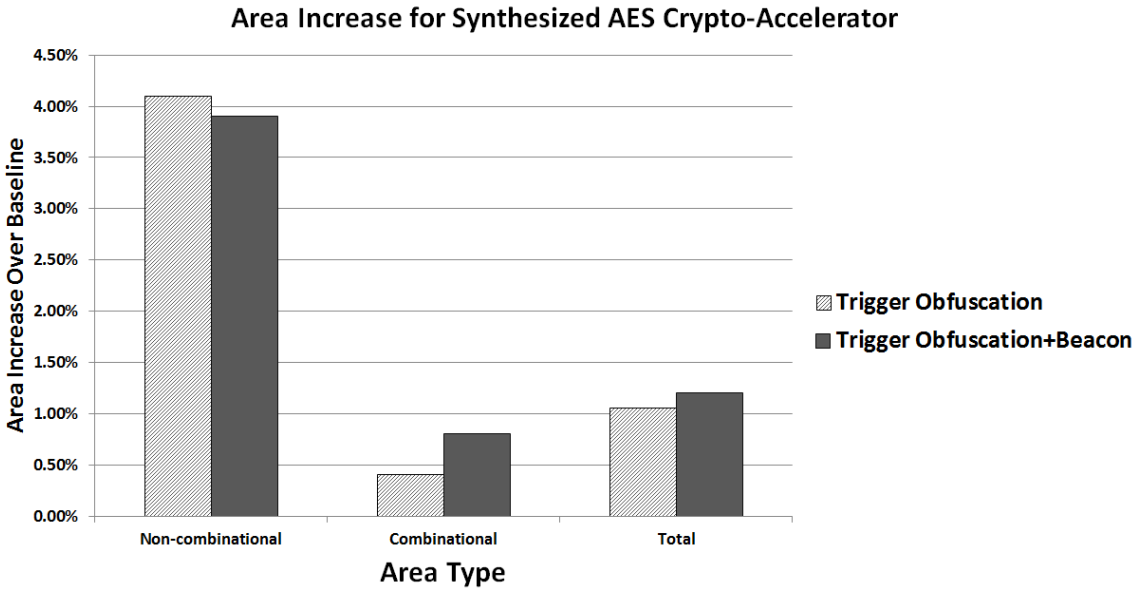

Figure 14.8: Area overheads of a trigger obfuscation defense mechanism and a beacon placed in a custom AES crypto-accelerator. Overheads are shown as a percentage of the baseline system. Timing requirements are maintained in all three designs.

to achieve. Specifically, if there is information one wants to hide about *some* circuits, even if it is only one bit of information, any obfuscated version of those circuits can be used to steal that information. A set of such unobfuscatable examples were constructed. Whether or not domain-specific obfuscation could be applied was left open. However, the set of unobfuscatable examples includes a set of constant-depth circuits, so even in the hardware setting a generally applicable, provably unbreakable obfuscation method can never be achieved. This work was later extended to show that for a family of programs, even approximate obfuscation is impossible, meaning that if complete obfuscation is the goal, even approximate computation is impossible [Bitansky and Paneth, 2013]. However, it remains possible to achieve approximate obfuscation while producing precise computation, which is exactly what our aim is with our development of beacons.

14.4 Generality and Applicability of Beacon Implementations

In our evaluation and case studies, we discussed various implementations of beacons. These implementations could be applied to any standard ASIC hardware design and could be adapted to custom circuit designs as well. In this section, we describe aspects of the beacon approach that allow for natural future extensions that could be developed and applied if they become necessary or financially motivated.

• Beacon circuitry can perform any common functionality, as long as it has a high activity factor. It does not need to have an activity factor of exactly one. This could be useful if it is desirable to make the beacon circuitry look like something specific, such as ALUs.

• Low-level synthesis attributes are configurable. For example, if an entire design is done with low drive strength, the beacon can be configured to have the same low drive strength. Other physical attributes, such as frequency and voltage, can be similarly matched.

• There are no restrictions on how intense or long-lasting a power side-channel beacon needs to be. Regardless of circumstantial details — such as the ability of security engineers to measure power emanations — beacons can be configured to match the needs of engineers.

Furthermore, the exact design restrictions on beacons impact the trade-offs between different approaches. For instance, in high-frequency designs, designers may want the beacon delay to be comparable with the length of the critical paths between latches. There are a few easy ways to solve this problem:

One option is to use a clock period divider so that the beacon signal takes multiple clock cycles (in the primary clock domain) to propagate. As a proof of concept, we implemented a version of this without problems using standard ASIC tools.

A second option is to use pipeline latches to make the beacon match the fastest clock domain. The only downside is that latches take up area and suffer from leakage power dissipation. This does not detract from the effectiveness of the beacon, but it increases the overhead by the cost of the latches.

A last option is to use unreliable latches for the beacon grid. This allows the beacon to meet any timing requirement. It is also a good place to use unreliable latches, because

correctness is irrelevant. Whether or not this is a viable option depends on the choice of foundry.

One last point of note relates to the choice of the work that the beacon does. If malicious foundries improve the state-of-the-art in the future, it might become possible to guess which gates are too quiet or are doing abnormal work. An optional trade-off then would be to do work in beacons that looks like real work. For example, a beacon could perform a series of addition operations, the results of which could be thrown away.

Part V

Conclusions and Future Research Directions

Chapter 15

Recommendations for Practical and Operational Security

We consider it important for security methodologies to be not only of academic interest but also to be useful and practical for real-world applications. Toward this end, we mention several of the practical aspects of our systems that have occurred to us when considering operational security and commercial viability.

• When applying design-side defenses, it is important to have a distinction between trusted and untrusted personnel. This seems especially important for military applications or applications in other regimes where a very limited number of personnel can be given complete trust. As is always the case in security, it is impossible to trust nobody. As we have discussed in this thesis, we are trusting the high level architectural specification. In other words, we trust that if we succeed in producing the hardware we are trying to build, we have built something that has desirable functionality. We also need to trust the person or people adding the security features, such as **TrustNet**, **DataWatch** and/or trigger obfuscation. These features need to be added after the design has been completed and have to be included in the system. We can imagine one person with a high level of clearance adding these small circuits or having a group of multiple people oversee and verify the addition. We note that these circuits are so simple and easily verifiable that as long as one trustworthy individual sees their inclusion, we can be assured that they have been correctly added, even

if several other malicious individuals have had the opportunity to tamper with the design.

- When performing static analysis, it is important to be clear on what guarantees are provided and what are not. Static analysis can be performed on source code, on gatelists, and possibly on netlists. Static analysis provides information only about the design phase it is being applied to. For example, if static analysis is applied to a gatelist and then a tool synthesizes the gatelist into a netlist, there is no guarantee that the netlist is backdoor-free, since the tool could have maliciously converted a benign gatelist into a faulty netlist. Static analysis provides security only for the phase during which it is applied.
- When implementing post-fabrication tests and/or audits, there is a degree of trust that is necessary. Allowing an untrusted foundry to audit its own chips seems problematic, as they could perform the audits incorrectly. For audits to be meaningful, they either need to be performed by a trusted person on site or performed on chips after they have been delivered to the purchasing company. Either way, in order for audits to have trustworthy results, they must be performed or overseen by a trustworthy individual (or automated to a degree that the results can be trusted by the relevant parties).
- Expectations of validation engineers can vary wildly from one scenario to another. In some cases, such as the incorporation of third-party IP into an otherwise trusted design, validation teams might be given complete trust and be expected to perform rigorous testing. In other scenarios, validation teams might be compromised or might perform only the most cursory of tests. We recommend that security engineers remain aware of the goings-on of validation teams and the implicit or explicit expectations. These expectations impact all aspects of hardware security. For example, the results of **FANCI** are only meaningful when combined with some degree of validation testing to catch frequent-action backdoors. Applications of power resets and other elements of trigger obfuscation can only provide guarantees when the module interfaces are appropriately validated and when architectural state is documented. In general, an awareness of the validation process and how it impacts the assumptions of various security methods is vital to operational security.

Chapter 16

Concluding Remarks

In this dissertation, we have presented a variety of methods for hardening modern hardware against the various threats that exist within the hardware development life cycle. We have also presented a holistic methodology for combining and applying all of these methods to create trustworthy hardware in the presence of a sophisticated and resourceful conspiracy of attackers. We conclude with a summary of the main contributions of this work, lessons that have been learned and limitations on this field of study moving forward.

16.1 Summary of Contributions

FANCI: We introduced a new method for using static analysis to detect potentially malicious circuitry within hardware designs as either source code or gatelists. Using this method, we were able to efficiently detect backdoors in modern benchmarks. We were also able to evaluate full microprocessor cores. We found that the scalability and generality of this approach are sufficient to make it a valuable tool for next generation hardware. We also conducted a red team/blue team exercise where international teams of hardware designers attempted to design stealthy backdoors that could evade detection.

Trigger Obfuscation: We proposed the first method for disabling unknown and undetected backdoors at runtime using trigger obfuscation. With this method, we are able to cut off hardware backdoor triggers before they turn on payloads, thus dynamically preventing attacks, even if we are unaware of the existence of the backdoors. We were able to efficiently

implement these methods on a large scale, both in an open source microprocessor (looking at the cores and the memory system) and in a custom microcontroller that we built from scratch with security in mind.

TrustNet and **DataWatch**: We proposed the first systems for dynamic monitoring of hardware behavior for backdoor payload detection. The two systems we built, **TrustNet** and **DataWatch**, allow for the detection of malicious payloads on the granularity of a single cycle (nanosecond scale). We were able to apply these systems with low simulated overheads to an open source microprocessor. We were further able to implement them efficiently into a homegrown cryptographic accelerator. These two systems both have demonstrated good scalability and avoid false positives by being implemented directly into the microarchitecture.

Beacons: We proposed beacons, an attestation mechanism for allowing the post-fabrication auditing of design-side protection mechanisms. This is the first hardware security system that protects both the design side and the fabrication side simultaneously and allows for a coherent framework with which to understand the full hardware development life cycle. We implemented these beacons at low cost into a variety of designs and applied entanglement methods to make reverse engineering difficult.

16.2 Lessons Learned

Full System Awareness and Defense In Depth: Working on these projects has convinced us that both full system awareness and the application of defense-in-depth techniques will remain important in hardware security fields. The nature of hardware development requires that several layers have to all work together. The work done in architectural design and coding is radically different than the manufacturing work done in a foundry. Most prior literature has focused only on one aspect of hardware development and understandably so. Moving forward, we believe that these communities could benefit from increase collaboration and cross-field research. For similar reasons, we believe defense in depth will become more valuable rather than less so. Even with better models and more advanced defenses, the variety of axioms and environments present in all the different possible ways hardware

can be developed will likely make it impossible for one simple technique to ever guarantee security from start to finish.

Trust Models: Various literature in these fields of research have significantly different trust models, and slight changes to the trust model can radically alter the way research is interpreted. From the specification through to the manufacturing, there are many different sets of personnel who might become involved, and these all change the way security is modeled. Expectations of what a validation team is capable of or likely to have funds for differ significantly. Similarly, perceptions of where the greatest and most immediate threats lie vary from researcher to researcher. Ideally, having a unified and static trust model (such as the one used in this dissertation) has value. However, we consider it unlikely that a single trust model will satisfy the global community for a long period of time. A factor that comes into play is the constant change in technology. As we move to lower technology nodes, different techniques are being used at foundries that could impact audits. Additionally, the usage of non-volatile memories and FPGAs impact the trust model and might become more prevalent in the future. Moves to other technologies entirely could change the tools that are used and/or design and validation practices. A challenge for the community is to maintain a coherent trust model in the future in the face of all of these changes.

Completeness of Axioms: A challenge that arises in any practical area of security is the shifting view of axioms. Generally, a security system guarantees security given its axioms and loses those guarantees if any of those axioms changes or becomes unrealistic. With hardware especially, axioms can change, or we can realize that our axioms caused us to miss out on something important. As we move to new architectures, including potentially massive amounts of accelerators, multiple clock domains, analog accelerators and a general increase in heterogeneity of hardware on chip, we have to think with each change how our axioms might change. As one specific example, can a new generation of accelerators have power resets applied? Do we understand how its interfaces might be attacked and what invariants we might want to enforce? Will the core functionality have an easily applicable homomorphic obfuscating function? Any time a substantially new hardware components becomes popular, these questions and more will have to be asked. We consider it an important challenge for the future to maintain a coherent view of hardware development as

the global economy and hardware needs continue to change.

16.3 Limitations and Necessary Future Directions

In security, there are always limitations. We discuss a few of the limitations of hardware security and touch on some future directions that we consider to be vital.

Arguably the greatest limitation on hardware security is personnel. Depending on the scenario, varying degrees of trusted personnel are necessary, but that number is never zero. It is possible that companies today feel that they can trust all personnel except for third parties and foundries. Whether or not that perspective is reasonable is currently a matter of opinion. Some elements of trust seem unavoidable, even in extreme military settings. The final post-fabrication audit requires some degree of trust due to the Principle of Last Action. Early steps in specification and architectural design seem to require a degree of trust as well. The methods proposed in this work drastically decrease the reliance on trusted personnel from where it is today. However, we do not believe that degree of trust can ever reach zero. One avenue of research that might help with this would be an open source library of hardware components, where this trust could be crowd-sourced to a degree. However, unless hardware technologies and microarchitectural techniques become more stagnant, it is unlikely that such a library would be efficient enough for commercial applications.

Another key limitation in the hardware space is that the product is ultimately physical. It seems potentially impossible to rule out all possible attacks. While this dissertation contributes greatly for digital attacks, the space of all possible attacks is larger and less well understood. Purely parametric attacks, which might not require any alterations to source code or even to digital gates, could exist in an almost limitless number of fashions. Is it possible that fine-grained temperature alterations could trigger unexpected events? Could physical environments be altered slightly by remote attackers in ways that fabrication engineers would not expect? Digital models and simulation environments are limited in precision. Thus, as long as security research takes place in simulated environments, there are limitations on the types of unknown attacks that might be prevented or discovered. We consider this limitation to be potentially fundamental, as physics continues to be a

developing and changing area, so protecting against all possible physical interference seems impossible.

In our opinion, the most vital direction for future research in hardware security is to understand the commercial viability of security approaches and improve efficiency where necessary. We know that modern hardware is vulnerable to attacks. However, modern hardware by and large is not protected by any security measures. It is possible that a large-scale commercial change will not occur until after a global catastrophe has occurred. Nevertheless, it is possible that commercial and/or government entities could act proactively and pursue security methods in the near future. We consider the biggest disincentive from a commercial perspective to be increased design complexity and delayed time to market. Even if area and power overheads are zero, designers will always shy away from complexity, as it increases design and validation costs. Additionally, any delay to time to market is a *de facto* area overhead, as technologies are always improving. Therefore, if security is to become a first-order concern in hardware development, it will likely need to become integrated into modern processes in such a way as to not become a source of delays and overheads. We anticipate that methods similar to the ones proposed in this thesis will become necessary in order to have reliable hardware in the future and hope that a proactive approach will be adopted.

Part VI

Bibliography

Bibliography

[Abdel-hamid *et al.*, 2003] Amr T. Abdel-hamid, Sofine Tahar, and El Mostapha Aboulhamid. IP Watermarking Techniques: Survey and Comparison. In *In IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. http://ieeexplore.ieee.org/Xplore/defdeny.jsp?url=/iel5/ 8609/27279/01213006.pdf*, 2003.

[Abdel-Hamid *et al.*, 2006] Amr T. Abdel-Hamid, Sofiène Tahar, and El Mostapha Aboulhamid. Finite State Machine IP Watermarking: A Tutorial. In *AHS*, pages 457–464, 2006.

[Aciicmez *et al.*, 2007a] O. Aciicmez, S. Gueron, and J. P. Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. Cryptology ePrint Archive, Report 2007/039, February 2007.

[Aciicmez *et al.*, 2007b] O. Aciicmez, C. K. Koc, and J. P. Sefert. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 312–320, March 2007.

[Aciicmez *et al.*, 2007c] O. Aciicmez, C. K. Koc, and J. P. Seifert. Predicting Secret Keys via Branch Prediction. In *Proceedings of the RSA Conference — Cryptographers Track (CT-RSA)*, pages 225–242, March 2007.

[Aciicmez *et al.*, 2007d] O. Aciicmez, W. Schindler, and C. K. Koc. Cache Based Remote Timing Attack on the AES. In *Proceedings of the RSA Conference — Cryptographers Track (CT-RSA)*, pages 271–286, March 2007.

[Aciicmez, 2007] O. Aciicmez. Yet Another MicroArchitectural Attack: Exploiting I-cache. In *Proceedings of the* 1^{st} *Computer Security Architecture Workshop (CSAW)*, pages 11–18, November 2007.

[Adee, 2008] Sally Adee. The Hunt for the Kill Switch. *IEEE Spectrum Magazine*, 45(5):34–39, 2008.

[Agosta *et al.*, 2007] G. Agosta, L. Breveglieri, I. Koren, G. Pelosi, and M. Sykora. Countermeasures Against Branch Target Buffer Attacks. In *Proceedings of the* 4^{th} *Workshop on Fault Diagnosis and Tolerance in Cryprography (FDTC)*, 2007.

[Aiello *et al.*, 1989] G. R. Aiello, M. Budinich, and E. Milotti. Hardware Implementation of a GFSR Pseudo-random Number Henerator. *Computer Physics Communications*, 56(2):135 – 139, 1989.

[Altschuler and Zoppis, 2008] Frank Altschuler and Bruno Zoppis. Embedded System Security. January 2008.

[ana,] Latest from DAC: ST and Media Tek manage media SoC designs (part 2). `http://www.edn.com/blog/1690000169/post/290028029.html`.

[Appenzeller, 1995] D. P. Appenzeller. Formal Verification of a PowerPC Microprocessor. In *ICCD '95: Proceedings of the 1995 International Conference on Computer Design*, page 79, Washington, DC, USA, 1995. IEEE Computer Society.

[Asonov and Agrawal, 2004] D. Asonov and R. Agrawal. Keyboard Acoustic Emanations. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 3–11, May 2004.

[Banga and Hsiao, 2008] Mainak Banga and Michael S. Hsiao. A Region Based Approach for the Identification of Hardware Trojans. In *Hardware-Oriented Security and Trust, 2008. HOST '08. IEEE International Workshop on*, June 2008.

[Banga *et al.*, 2008] Mainak Banga, Maheshwar Chandrasekar, Lei Fang, and Michael S. Hsiao. Guided Test Generation for Isolation and Detection of Embedded Trojans in ICs. In *GLSVLSI '08: Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 363–366, New York, NY, USA, 2008. ACM.

[Barak *et al.*, 2001] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 1–18, London, UK, UK, 2001. Springer-Verlag.

[Becker *et al.*, 2010] G.T. Becker, M. Kasper, A. Moradi, and C. Paar. Side-channel Based Watermarks for Integrated Circuits. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*, pages 30 –35, june 2010.

[Becker *et al.*, 2013] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy Dopant-Level Hardware Trojans. In *CHES*, pages 197–214, 2013.

[Bellare *et al.*, 2012] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of Garbled Circuits. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, CCS '12, pages 784–796, New York, NY, USA, 2012. ACM.

[Bernstein, 2005] D. J. Bernstein. Cache-timing Attacks on AES, 2005.

[Biham *et al.*, 2008] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug Attacks. In *CRYPTO*, pages 221–240, 2008.

[Bitansky and Paneth, 2013] Nir Bitansky and Omer Paneth. On the Impossibility of Approximate Obfuscation and Applications to Resettable Cryptography. In *STOC*, pages 241–250, 2013.

[Bonneau and Mironov, 2006] J. Bonneau and I. Mironov. Cache-Collision Timing Attacks Against AES. In *Proceedings of the* 8^{th} *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 201–215, 2006.

[Brickell *et al.*, 2006] E. Brickell, G. Graunke, M. Neve, and J. P. Seifert. Software Mitigations to Hedge AES Against Cache-based software side channel vulnerabilities. IACR ePrint Archive, Report 2006/052, February 2006.

[Brzozowski and Yarmolik, 2007] Maciej Brzozowski and Vyacheslav N. Yarmolik. Obfuscation as Intellectual Rights Protection in VHDL Language. In *Proceedings of the 6th*

International Conference on Computer Information Systems and Industrial Management Applications, CISIM '07, pages 337–340, Washington, DC, USA, 2007. IEEE Computer Society.

[Carretero *et al.*, 2009] Javier Carretero, Pedro Chaparro, Xavier Vera, Jaume Abella, and Antonio González. End-to-end Register Data-flow Continuous Self-test. *SIGARCH Comput. Archit. News*, 37(3):105–115, 2009.

[Chakraborty and Bhunia, 2008] Rajat Subhra Chakraborty and Swarup Bhunia. Hardware protection and authentication through netlist level obfuscation. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '08, pages 674–677, Piscataway, NJ, USA, 2008. IEEE Press.

[Chakraborty and Bhunia, 2009] Rajat Subhra Chakraborty and Swarup Bhunia. Security through Obscurity: An approach for Protecting Register Transfer Level Hardware IP. In *Proceedings of the 2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, HST '09, pages 96–99, Washington, DC, USA, 2009. IEEE Computer Society.

[Chakraborty *et al.*, 2008] R.S. Chakraborty, S. Paul, and S. Bhunia. On-Demand Transparency for Improving Hardware Trojan Detectability. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 48–50, June 2008.

[Chapman and Durrani, 2000] R. Chapman and T.S. Durrani. IP Protection of DSP Algorithms for System on Chip Implementation. *Signal Processing, IEEE Transactions on*, 48(3):854 –861, mar 2000.

[Chapman *et al.*, 1999] R. Chapman, T.S. Durrani, and A. P. Tarbert. Watermarking DSP Algorithms for System on Chip Implementation. In *Electronics, Circuits and Systems, 1999. Proceedings of ICECS '99. The 6th IEEE International Conference on*, volume 1, pages 377–380 vol.1, 1999.

[Chatterjee *et al.*, 2000] Saugata Chatterjee, Chris Weaver, and Todd Austin. Efficient Checker Processor Design. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 87–97, New York, NY, USA, 2000. ACM.

[Choudhary *et al.*, 2011] N.K. Choudhary, S.V. Wadhavkar, T.A. Shah, H. Mayukh, J. Gandhi, B.H. Dwiel, S. Navada, H.H. Najaf-abadi, and E. Rotenberg. Fabscalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 11–22. IEEE, 2011.

[Coron, 1999] J. Coron. Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems. In C .K. Koc and C. Paar, editors, *Proceedings of the* 1^{st} *Cryptographic Hardware and Embedded Systems*, pages 292–302, August 1999.

[Dyer *et al.*, 2001] J.G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S.W. Smith. Building the IBM 4758 Secure Coprocessor. *Computer*, 34(10):57–66, Oct 2001.

[edn, 2008] Intel's Silverthorne Unveiled: Detailing Baby Centrino. `http://www.anandtech.com/showdoc.aspx?i=3230&p=4`, 2008.

[Elbaz *et al.*, 2009] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B. Lee, Nachiketh Potlapally, and Lionel Torres. Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines. pages 1–22, 2009.

[esc, 2013] The 2013 Embedded Systems Challenge. 2013.

[Gandolfi *et al.*, 2001] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In *Proceedings of* 3^{rd} *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 251–261, 2001.

[Gassend *et al.*, 2002] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *ACM Conference on Computer and Communications Security*, pages 148–160, New York, NY, USA, 2002. ACM Press.

[Gentry, 2010] Craig Gentry. Computing Arbitrary Functions of Encrypted Data. *Commun. ACM*, 53(3):97–105, 2010.

[Goering, 2003] Richard Goering. Software Engineering Firm Obfuscates Verilog, 2003.

[Good and Benaissa, 2005] Tim Good and Mohammed Benaissa. AES on FPGA from the Fastest to the Smallest. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop*, pages 427–440. Springer, 2005.

[Harada *et al.*, 1997] T. Harada, H. Sasaki, and Y. Kami. Investigation on Radiated Emission Characteristics of Multilayer Printed Circuits Boards. *IEICE Transactions on Communications*, E80-B(11):1645–1651, 1997.

[Helfmeier *et al.*, 2013] Clemens Helfmeier, Dmitry Nedospasov, Christopher Tarnovsky, Jan Krissler, Christian Boit, and Seifert Jean-Pierre. Breaking and Entering Through the Silicon. In *Proceedings of the 2013 ACM conference on Computer and Communications Security*, CCS '13, 2013.

[Hicks *et al.*, 2010] Matthew Hicks, Samuel T. King, Milo M. K. Martin, and Jonathan M. Smith. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[IBM,] IBM. IBM 4764 PCI-X Cryptographic Coprocessor.

[ITR,] International Technology Roadmap for Semiconductors 2009 Edition: Executive Summary.

[Jin and Makris, 2008] Yier Jin and Yiorgos Makris. Hardware Trojan Detection Using Path Delay Fingerpring. In *Hardware-Oriented Security and Trust, 2008. HOST '08. IEEE International Workshop on*, June 2008.

[Kahng *et al.*, 2001] Andrew B. Kahng, John Lach, William H. Mangione-smith, Stefanus Mantik, Student Member, Igor L. Markov, Miodrag Potkonjak, Paul Tucker, Huijuan Wang, and Gregory Wolfe. Constraint-based Watermarking Techniques for Design IP Protection. *IEEE Trans. Computer-Aided Design Integrated Circuits Systems*, 20:1236–1252, 2001.

[King *et al.*, 2008] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and Implementing Malicious Hardware. In *Proceedings of the* 1^{st} *USENIX Workshop on Large-scale Exploits and Emergent Threats*, April 2008.

[Knuth, 1981] Donald E. Knuth. *Seminumerical Algorithms, The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, Mass., USA, 1981.

[Kocher *et al.*, 1999] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. pages 388–397. Springer-Verlag, 1999.

[Kömmerling and Kuhn, 1999] Oliver Kömmerling and Markus G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 9–20, May 1999.

[Koushanfar and Alkabani, 2010] Farinaz Koushanfar and Yousra Alkabani. Provably Secure Obfuscation of Diverse Watermarks for Sequential Circuits. In *HOST*, pages 42–47, 2010.

[Lampson, 1973] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10), 1973.

[Lee *et al.*, 2004] Jae W. Lee, Daihyun Lim, Blaise Gassend, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. A Technique to Build a Secret Key in Integrated Circuits for Identification and Authentication Application. In *Proceedings of the Symposium on VLSI Circuits*, pages 176–159, 2004.

[len, 2006] U.S. Government Restricts China PCs, 2006.

[Li and Lach, 2008] Jie Li and J. Lach. At-Speed Delay Characterization for IC Authentication and Trojan Horse Detection. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 8–14, June 2008.

[Liu *et al.*, 2008] Chao-Liang Liu, Gwoboa Horng, and Hsin-Yu Liu. Computing the Modular Inverses is as Simple as Computing the GCDs. *Finite Fields and Their Applications*, 14(1):65–75, 2008.

[Lu and Tseng, 2002] Chih Chung Lu and Shau Yin Tseng. Integrated Design of AES Encrypter and Decrypter. In *Application-Specific Systems, Architectures and Processors, 2002. Proceedings. The IEEE International Conference on*, pages 277 – 285, 2002.

[Mangard *et al.*, 2007] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, Secaucus, NJ, USA, 2007.

[Mangard, 2003] S. Mangard. Exploiting Radiated Emissions - EM Attacks on Cryptographic ICs. In *Proceedings of AustroChip*, 2003.

[Marchetti and Marks, 1974] V. Marchetti and J. Marks. *The CIA and the Cult of Intelligence*. Knopf, 1974.

[Marsaglia, 2003] George Marsaglia. Random Number Generators. *Journal of Modern Applied Statistical Methods*, 2(1):2–13, 2003.

[Matsumoto and Nishimura, 1998] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998.

[Mulder *et al.*, 2005] E. De Mulder, P. Buysschaert, S. B. Ors, P. Delmotte, B. Preneel, G. Vandenbosch, and I. Verbauwhede. Electromagnetic Analysis Attack on an FPGA Implementation of an Elliptic Curve Cryptosystem. In *Proceedings of EUROCON*, November 2005.

[Narasimhan *et al.*, 2012] S. Narasimhan, W. Yueh, X. Wang, S. Mukhopadhyay, and S. Bhunia. Improving IC Security against Trojan Attacks through Integration of Security Monitors. *Design Test of Computers, IEEE*, PP(99):1, 2012.

[Network, 2012] Intel Software Network. Intel Advanced Encryption Standard (AES) Instructions Set - Rev 3, 2012.

[Neve and Seifert, 2006] M. Neve and J. P. Seifert. Advances on Access-driven Cache Attacks on AES. In *Proceedings of Selected Areas of Cryptography (SAC)*, 2006.

[Neve *et al.*, 2006] M. Neve, J. P. Sefert, and Z. Wang. A Refined Look at Bernstein's AES Side-channel Analysis. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, page 369, March 2006.

[Osvik *et al.*,] D. A. Osvik, A. Shamir, and E. Tromer. Other People's Cache: Hyper Attacks on HyperThreaded Processors. Presentation available at `http://www.wisdom.weizmann.il/~tromer/`.

[Osvik *et al.*, 2005] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and Countermeasures: the Case of AES. Cryptology ePrint Archive, Report 2005/271, 2005.

[Park and Miller, 1988] S. K. Park and K. W. Miller. Random Number Generators: Good Ones are Hard to Find. *Commun. ACM*, 31:1192–1201, October 1988.

[Percival,] C. Percival. Cache Missing for Fun and Profit. `http://www.daemonology.net/papers/htt.pdf`.

[Quisquater and Samyde, 2001] J. J. Quisquater and D. Samyde. Electromagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *Proceedings of the International Conference on Smart Cards: Smart Card Programming and Security (E-smart)*, pages 200–210, 2001.

[Rad *et al.*, 2008] Reza M. Rad, Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. Power Supply Signal Calibration Techniques for Improving Detection Resolution to Hardware Trojans. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 632–639, Piscataway, NJ, USA, 2008. IEEE Press.

[Rajendran *et al.*, 2012] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Security Analysis of Logic Obfuscation. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 83–89, New York, NY, USA, 2012. ACM.

[Rajendran *et al.*, 2013] Jeyavijayan Rajendran, Michael Sam, Ozgur Sinanoglu, and Ramesh Karri. Security Analysis of Integrated Circuit Camouflaging. In *Proceedings of*

the 20th ACM Conference on Computer and Communications Security, CCS '13. ACM, 2013.

[Reinhardt and Mukherjee, 2000] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36, New York, NY, USA, 2000. ACM.

[Rosenfeld and Karri, 2010] Kurt Rosenfeld and Ramesh Karri. Attacks and Defenses for JTAG. *Design & Test of Computers, IEEE*, 27(1):36–47, Jan.-Feb. 2010.

[Roy *et al.*, 2008] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. EPIC: Ending Piracy of Integrated Circuits. *Design, Automation & Test in Europe Conference & Exhibition*, 0:1069–1074, 2008.

[Salmani *et al.*, 2009] H. Salmani, M. Tehranipoor, and J. Plusquellic. New Design Strategy for Improving Hardware Trojan Detection and Reducing Trojan Activation Time. In *Hardware-Oriented Security and Trust, 2009. HOST '09. IEEE International Workshop on*, pages 66–73, July 2009.

[Saputra *et al.*, 2003] H. Saputra, N. Vijaykrishnan, M. Kandemir, M. Irwin, R. Brooks, S. Kim, and W. Zhang. Masking the Energy Behavior of DES Encryption. In *Proceedings of the Design Automation and Test in Europe Conference (DATE)*, 2003.

[Shamir and Tromer,] A. Shamir and E. Tromer. Acoustic Cryptanalysis: On Nosy People and Noisy Machines. `http://people.csail.mit.edu/tromer/acoustic/`.

[Simonite, 2013] Tom Simonite. *NSA's Own Hardware Backdoors May Still Be a Problem From Hell.* MIT Technology Review, October 2013.

[Smith, 2004] Sean Smith. Magic Boxes and Boots: Security in Hardware. *IEEE Computer*, 37(10):106–109, 2004.

[Sturton *et al.*, 2011] Cynthia Sturton, Matthew Hicks, David Wagner, and Samuel T. King. Defeating UCI: Building Stealthy and Malicious Hardware. In *Proceedings of*

the 2011 IEEE Symposium on Security and Privacy, SP '11, pages 64–77, Washington, DC, USA, 2011. IEEE Computer Society.

[Suh and Devadas, 2007] G. Edward Suh and Srinivas Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *Design Automation Conference*, pages 9–14, New York, NY, USA, 2007. ACM Press.

[Synopsys, 2006] Synopsys. Design Compiler Technology Backgrounder, 2006.

[tcg, 2007] Trusted Computing Group. Online at `https://www.trustedcomputinggroup.org/`, 2007.

[Tehranipoor *et al.*, 2012] Mohammad Tehranipoor, Ramesh Karri, Farinaz Koushanfar, and Miodrag Potkonjak. TrustHub, 2012.

[Thompson, 1984] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, August 1984.

[Tiri and Verbauwhede, 2005a] K. Tiri and I. Verbauwhede. A VLSI Design Flow for Secure Side-Channel Attack Resistant ICs. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 58–63, March 2005.

[Tiri and Verbauwhede, 2005b] K. Tiri and I. Verbauwhede. Design Method for Constant Power Consumption of Differential Logic Circuits. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, pages 628–633, March 2005.

[Tiri and Verbauwhede, 2006] K. Tiri and Ingrid Verbauwhede. A Digital Design Flow for Secure Integrated Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(7):1197–1208, July 2006.

[Tiri *et al.*, 2007] K. Tiri, O. Aciicmez, M. Neve, and F. Andersen. An Analytical Model for Time-Driven Cache Attacks. In *Proceedings of the Fast Software Encryption Workshop (FSE)*, March 2007.

[Uni, 2005] United Stated Department of Defense. *High Performance Microchip Supply*, February 2005.

[Valamehr *et al.*, 2012] Jonathan Valamehr, Melissa Chase, Seny Kamara, Andrew Putnam, Dan Shumow, Vinod Vaikuntanathan, and Timothy Sherwood. Inspection Resistant Memory: Architectural Support for Security from Physical Examination. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 130–141, Washington, DC, USA, 2012. IEEE Computer Society.

[Verbauwhede *et al.*, 2006] I. Verbauwhede, K. Tiri, D. Hwang, A. Hodjat, and P. Schaumont. Circuits and Design Techniques for Secure ICs Resistant to Side-Channel Attacks. In *Proceedings of the International Conference on IC Design & Technology (ICICDT)*, pages 1–4, May 2006.

[Verilog, 1991] Verilog. Verilog-HDL PLI Reference Manual, 1991.

[von Neumann, 1956] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, pages 43–99, 1956.

[Waksman and Sethumadhavan, 2010] Adam Waksman and Simha Sethumadhavan. Tamper Evident Microprocessors. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, Oakland, California, 2010.

[Waksman and Sethumadhavan, 2011] Adam Waksman and Simha Sethumadhavan. Silencing Hardware Backdoors. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 49–63, Washington, DC, USA, 2011. IEEE Computer Society.

[Waksman *et al.*, 2013a] A. Waksman, J. Eum, and S. Sethumadhavan. Practical, Lightweight Secure Inclusion of Third-Party Intellectual Property. In *Design and Test, IEEE*, 2013.

[Waksman *et al.*, 2013b] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13. ACM, 2013.

[Waksman *et al.*, 2014] Adam Waksman, Jeyavijayan Rajendran, Matthew Suozzo, and Simha Sethumadhavan. A Red Team/Blue Team Assessment of Functional Analysis

Methods for Malicious Circuit Identification. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, 2014.

[Wang *et al.*, 2008] Xiaoxiao Wang, M. Tehranipoor, and J. Plusquellic. Detecting Malicious Inclusions in Secure Hardware: Challenges and Solutions. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 15–19, June 2008.

[Yoo and Franklin, 2008] Joonhyuk Yoo and Manoj Franklin. Hierarchical Verification for Increasing Performance in Reliable Processors. *J. Electron. Test.*, 24(1-3):117–128, 2008.

[Yu and Devadas, 2010] Meng-Day (Mandel) Yu and Srinivas Devadas. Secure and Robust Error Correction for Physical Unclonable Functions. *Design & Test of Computers, IEEE*, 27(1):48–65, Jan.-Feb. 2010.

Part VII

Appendices

Appendix A

Glossary of Terms

Backdoor: An alteration to hardware that allows it to violate the ISA contract, generally assumed to be both malicious and surreptitious.

Backdoor Payload: The result (often as a digital emission) of a backdoor, *i.e.,* the ends being achieved by a backdoor.

Backdoor Trigger: The signal (digital or otherwise) that causes backdoor behavior to initiate.

Beacon: A hardware circuit that creates a measurable but undocumented output when supplied with a specific input (such as a key).

Cache: A (usually small) memory that stores recently accessed data for the purpose of speeding up common-case memory accesses.

Computer Architecture: The defined mechanisms and operations of a hardware system, usually specified in an ISA.

Control Corrupter Backdoor Payload: A backdoor payload wherein the payload action is achieved by altering control signals in pre-existing operations (such as changing an addition into a subtraction).

Dark Silicon: A phrase for describing the emergent phenomenon that in many modern chips only a fraction of the chip can operate at any given point in time as a result of fixed power budgets.

Data Corrupter Backdoor Payload: A backdoor payload wherein the payload action is achieved by causing an incorrect data operation (such as making it so that $5 + 6 = 12$).

Data Obfuscation: A technique for intentionally obfuscating data inputs to hardware modules to prevent the actions of backdoors triggered by single-shot cheat codes.

DataWatch: A hardware-based, self-monitoring system for dynamically detecting corrupter backdoor payloads.

Denial of Service: An attack wherein the goal is only to prevent a system from working, rather than to achieve any more specific malicious end.

Emitter Backdoor Payload: A backdoor payload wherein the payload action is contained within a superfluous emission that is supplementary to the normal microarchitectural traffic of the running program(s).

Fabrication: The process of manufacturing a physical computing chip.

FANCI: A static analysis algorithm and corresponding prototype for detecting stealthy circuits in hardware designs.

Foundry: A facility where hardware designs are used to manufacture physical chips.

Frequent-Action Backdoor: A backdoor that is frequently or always active and producing a payload.

Gatelist: An intermediate soft representation of hardware that specifies all gates in a design and how they are connected.

Hardware Design Language: A language for specifying the functionality of a hardware design that can be compiled into a netlist.

Instruction Set Architecture: Commonly referred to as an ISA, the document that specifies the computer architecture.

Main Channel: A defined channel through which information is meant to be transmitted, such as a defined interface.

Microarchitecture: The implementation details of hardware that operate at a lower level than the computer architecture (as specified by the ISA). Microarchitectural information is generally not needed to understand the architecture and generally impacts performance rather than correctness.

Netlist: The lowest level soft representation of hardware, describing connectivity information, transistor layouts and various physical attributes.

Netlist Entanglement: A form of netlist-level obfuscation for making it difficult to detect

the boundaries between semantically distinct chip elements.

Obfuscation: The act of intentionally making something hard to understand, usually with the connotation of the method not being cryptographically perfect.

Pathological Pipeline: An attack strategy against FANCI wherein designs are pipelined to an extreme and unnecessary degree, possibly including unnecessary feedback loops, so as to make the analysis of combinational logic difficult.

Principle of Last Action: The idea that given two sophisticated adversaries and unlimited resources, the actor who acts last usually wins.

Rapid Resets: A technique of rapidly reseting power and microarchitectural state in a hardware module to prevent the actions of backdoors triggered by ticking timebombs.

Register Transfer Level: A hardware design abstraction, mainly for synchronous digital circuits, that models circuit operations as the flow of digital signals between registers.

Sequence Cheat Code: A digital backdoor trigger that requires a series of data spread out across multiple clock cycles or events.

Sequence Reordering: A technique for reordering microarchitectural transactions benignly to prevent the actions of backdoors triggered by sequence cheat codes.

Single-Shot Cheat Code: A digital backdoor trigger that requires one data input that arrives at one point in time (or during a single clock cycle).

Side Channel: A channel through which information can be transmitted that exists due to implementation details rather than original design, often but not always existing by accident.

Stealthy Backdoor: A hardware backdoor whose operation is unlikely to be noticed during validation tests.

Technology Library: Documentation for the layouts of a set of implementations of logical gates and connections, often used in the compilation of hardware designs.

Technology Node: A generation of hardware technology, most commonly identified by the defining feature length (such as 32 nanometers).

Ticking Timebomb: A digital backdoor trigger that requires only timing information, such as the passage of clock cycles or a certain number of regular events.

Trigger Obfuscation: A term for the set of methods used to obfuscate hardware module

inputs so as to prevent backdoors from receiving digital triggers.

Trojan: An alternate term for a backdoor, sometimes with the connotation of having been implemented by a malicious foundry.

TrustNet: A hardware-based, self-monitoring system for dynamically detecting emitter backdoor payloads.

Validation: The term for general testing methods applied to hardware designs to test for reliability, correctness and/or security.

Verification: The process of proving properties about a hardware design, usually regarding correctness.