ABSTRACT

Title of dissertation: HARDWARE ATTACKS AND MITIGATION TECHNIQUES

Chongxi Bao, Doctor of Philosophy, 2017

Dissertation directed by: Professor Ankur Srivastava
Department of Electrical and
Computer Engineering

Today, electronic devices have been widely deployed in our daily lives, basic infrastructure such as financial and communication systems, and military systems. Over the past decade, there have been a growing number of threats against them, posing great danger on these systems. Hardware-based countermeasures offer a low-performance overhead for building secure systems. In this work, we investigate what hardware-based attacks are possible against modern computers and electronic devices. We then explore several design and verification techniques to enhance hardware security with primary focus on two areas: hardware Trojans and side-channel attacks.

Hardware Trojans are malicious modifications to the original integrated circuits (ICs). Due to the trend of outsourcing designs to foundries overseas, the threat of hardware Trojans is increasing. Researchers have proposed numerous detection methods, which either take place at test-time or monitor the IC for unexpected behavior at run-time. Most of these methods require the possession of a Trojan-free

IC, which is hard to obtain. In this work, we propose an innovative way to detect Trojans using reverse-engineering. Our method eliminates the need for a Trojan-free IC. In addition, it avoids the costly and error-prone steps in the reverse-engineering process and achieves significantly good detection accuracy. We also notice that in the current literature, very little effort has been made to design-time strategies that help to make test-time or run-time detection of Trojans easier. To address this issue, we develop techniques that can improve the sensitivity of designs to test-time detection approaches. Experiments show that using our method, we could detect a lot more Trojans with very small power/area overhead and no timing violations.

Side-channel attack (SCA) is another form of hardware attack in which the adversary measures some side-channel information such as power, temperature, timing, etc. and deduces some critical information about the underlying system. We first investigate countermeasures for timing SCAs on cache. These attacks have been demonstrated to be able to successfully break many widely-used modern ciphers. Existing hardware countermeasures usually have heavy performance overhead. We innovatively apply 3D integration techniques to solve the problem. We investigate the implication of 3D integration on timing SCAs on cache and propose several countermeasures that utilize 3D integration techniques. Experimental results show that our countermeasures increase system security significantly while still achieving some performance gain over a 2D baseline system. We also investigate the security of Oblivious RAM (ORAM), which is a newly proposed hardware primitive to hide memory access patterns. We demonstrate both through simulations and on FPGA board that timing SCAs can break many ORAM protocols. Some general guidelines

in secure ORAM implementations are also provided. We hope that our findings will motivate a new line of research in making ORAMs more secure.

HARDWARE ATTACKS AND MITIGATION TECHNIQUES

by

Chongxi Bao

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

Advisory Committee:
Professor Ankur Srivastava, Chair/Advisor
Professor Gang Qu
Professor Rajeev Barua
Professor Dana Dachman-Soled
Professor Jeffrey S. Foster

Dedication

To my parents.

Acknowledgments

I owe my gratitude to all the people, without whom this thesis would not be possible, and because of whom my PhD experience has been so unforgettable.

First and foremost I'd like to thank my advisor, Professor Ankur Srivastava for his unwavering support throughout my PhD life, for giving me an invaluable opportunity to work on challenging and extremely interesting projects, and for his guidance and patience over the past five years. He has always directed me to the right path when I was unsure of how to proceed. Without his invaluable advice and constant inspiration, I would not have been able to improve and achieve all that I have. It has been truly a pleasure to work with and learn from him.

I would also like to extend my gratitude to other members of my dissertation committee, Professor Rajeev Barua, Professor Gang Qu, Professor Dana Dachman-Soled, and Professor Jeff Foster, for their time and service.

My colleagues in Professor Srivastava's research group have enriched my graduate life in many ways and deserve a special mention as well. Domenic Forte gave me invaluable guidance on how to write papers, develop ideas and setup experiments during my first year. Bing Shi, Caleb Serafy and Tiantao Lu set good examples for me of how to behave well in the PhD career. I also owe my gratitude to Zhiyuan Yang, Yang Xie, Ankit Mondal, Yuntao Liu, and Abhishek Chakraborty for not only fruitful discussions on research topics, but also stress relieving jokes we shared in our lab.

I would also like to acknowledge the financial support provided by the ECE

department and Ann G. Wylie fellowship. Because of the financial security they provided, I could focus entirely on my research.

Last, but by no means least, I owe my deepest thanks to my family - my mother and father who pulled me through every hardship and Linna Zhang, who always stood by me. Their support, encouragement and advice are so precious that words cannot express the gratitude I owe them.

Table of Contents

List of Figures

List of Abbreviations

AES	Advanced Encryption Standard
BEOL	Back-End of Line
BPA	Branch Prediction Attack
BTB	Branch Target Buffer
CAD	Computer Aided Design
CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit
DMS	Data Migration Scheme
DoS	Denial of Service
EDA	Electronic Design Automation
FEOL	Front-End of Line
FPGA	Field Programmable Gate Array
FPR	False-Positive Rate
FSM	Finite State Machine
HT	Hardware Trojan
IC	Integrated Circuit
ICUT	IC Under Test
IP	Intellectual Property
IPC	Instruction Per Cycle
ISA	Instruction Set Architecture
LLC	Last Level of Cache
NUCA	Non-Uniform Cache Architecture
ORAM	Oblivious RAM
RAM	Random Access Memory
RE	Reverse-Engineering
RNG	Random Number Generator
RTL	Register Transfer Level
SAM	Scanning Acoustic Microsccopy
SCA	Side-Channel Attack
SDM	Steepest Descent Method
SEM	Scanning Electron Microscopy
SVF	Side-channel Vulnerability Factor
SVM	Support Vector Machine
TSV	Through-Silicon Via
TA	Trojan-Addition
TD	Trojan-Deletion
TF	Trojan-Free
TI	Trojan-Inserted
TMR	Trojan Miss Rate
TP	Trojan-Parametric

Chapter 1: Introduction

1.1 Why Do We Need Cyber Security?

With the help of continued device scaling and emerging technologies such as 3D-integration, the performance of electronic devices including personal computers and smartphones have been increasing rapidly in the past twenty years. For example, the supercomputer Cray-2 developed in 1985 can only perform 2 billion floating point operations per second, while a smartphone produced in 2015 can easily perform over 100 billion floating point operations per second. The dramatic increase in computing power allows these electronic devices to be used for a wide range of applications, including communication, banking, process control, etc. They also become the critical part of larger systems used in financial, commercial and military sectors.

As these electronic devices become more ubiquitous, they also become extremely vulnerable to various attacks. For example, the demand for high-performance has resulted in the invention of many computer components, such as branch prediction unit, cache, etc. They are used to fill in the speed gap between the fast CPU and the slow memory. However, they also introduce another level of vulnerability. Recently, researchers have proposed several attacks on these components that can successfully break many modern encryption algorithms running on the computer

system [1–5]. Moreover, the demand of lower time-to-market and lower price has driven big electronic companies to outsource their designs to be fabricated overseas. This makes it easier for untrusted third parties to make modifications of the original circuit, steal intellectual property, make counterfeits, or overproduce the products and sell them for profits [6–8].

Given the pervasiveness of computing devices in commercial and military systems, the above-mentioned attacks can have devastating consequences. For example, the attacks on caches can be used to break Advanced Encryption Standard (AES). AES is currently the most widely-used encryption algorithm. Breaking it essentially means that military communications can be eavesdropped, bank transactions can be forged, etc. Another example is the malicious modification of the original circuit, also known as hardware Trojans (HTs). HTs are inevitable in the current trend of outsourcing designs to foundry overseas. They can be used to change the functionality, cause the denial of service, or leak valuable information of the underlying system. Due to the decreasing size of device features and increasing complexity of modern systems, HTs are very hard to be detected. Consider what will happen if they evade detection and are placed in military systems or safety-critical systems (power grid, first response system, etc.).

Because of the ever growing number of possible attacks and potential devastating consequences, there is an urgent need for a comprehensive and effective way to enhance the integrity, reliability and security of electronic devices, private data, and hardware systems.

1.2 Possible Threats on Hardware Systems

In this Section, we will introduce possible threats on hardware systems. Typically, the life-time of a hardware system can be divided into the four phases:

- Design Time. In this phase, designers come up with the register transfer level (RTL) design of the underlying system. The RTL design then goes through synthesis, technology mapping, placement and routing steps and is converted to layouts that will be fabricated in foundries. The software running on the hardware system is also developed in this phase.

- Fabrication Time. In this phase, the layout is sent to one or multiple foundries to be fabricated. Because of economical reasons, most design companies do not keep an in-house foundry. Instead, they send layouts to foundries not in their control.

- Test Time. If multiple foundries are involved in the fabrication phase, then it is in this phase that multiple parts are finally assembled. Then the fabricated chips are inspected and tested for possible failure. This step also includes the testing of software running on the hardware system.

- Post-deployment. In this phase, the tested chips are deployed and perform their specified tasks.

We will introduce the threats that may take place during all of the above four phases next.

1.2.1 Threats during Design Time

During Design time, designers come up with the RTL design of the underlying system. They also write software that runs on the system if necessary. It is possible that a rogue employee in the design team may try to maliciously modify the design file or steal the whole design. Moreover, nowadays, most designers do not build the entire RTL design from scratch. Instead, they typically purchase some intellectual property (IP) from third parties and integrate them with their own IPs. For example, most likely, the designer of a smartphone will purchase the WiFi module from a third party and integrate it with his/her own modules. Although this practice effectively lowers the time-to-market, it also allows a rogue employee from IP vendors to insert malicious circuitry into the IP that causes malfunction or denial-of-service.

Another source of threat comes from the Electronic Design Automation (EDA) tools or compilers. In the design phase, RTL designs are synthesized, mapped, and then placed and routed to make layouts using EDA tools. If these tools are untrusted, then the quality of the resulting layouts is in question.

We can summarize the two major threats that may happen during design phase as follows:

- Hardware Trojans and Software Trojans. As stated above, a rogue employee in the design team or from IP vendors may maliciously modify the RTL design. This is known as hardware Trojans (HTs). EDA tools may also insert HTs. Similarly, if there is software development involved, then software Trojans may be inserted.

- IP or Software Piracy. Rogue employees in the design team can also steal the IP or software source code and later use them to make counterfeit instances. Note that the stolen IP or software can also be used to aid attacks that happen in a later phase.

1.2.2 Threats during Fabrication Time

In this phase, the layouts are sent to a foundry or multiple foundries to be fabricated. For economical reasons, most designers do not keep an in-house foundry. Instead, they outsource their designs to other foundries. Since the designers now lose control over the fabrication process and the foundries might not be trusted, three attacks might take place:

- Hardware Trojans. Adversaries in the foundry may modify the mask used in lithography. This allows them to insert HTs into the layout. These HTs will cause malicious behavior to the products.

- IC Piracy. Adversaries in the foundry may steal the layout entirely. The stolen layout can be reverse-engineered to gain knowledge about the RTL design. This knowledge may be used to make counterfeit products. It may also enable attacks that happen in the later phase.

- Overproduction. Once the foundry has the layout, it can overproduce the products and sell the additional copies for profit. This leads to the loss of profit in the product designer.

1.2.3 Threats during Test Time

If the RTL design is fabricated at multiple foundries, it is assembled during this phase. Then test vectors are applied to detect any unwanted or incorrect behavior. Two possible threats exist at this time.

- Hardware Trojans. During test time, attacker may choose to activate HTs already inserted during previous phase(s). It is also possible that a rogue employee in the testing team may deliberately ignore the incorrect behavior caused by the HTs in the design and thus leaving HTs active.

- Denial-of-Service (DoS). Because no testing will happen after the test phase, an attacker may choose to launch DoS attack at this time. When the altered products are deployed, fixing them will cause a lot of time and efforts from the designer, which is a huge loss of profit for the designer.

1.2.4 Threats during Post-Deployment

After the electronic device has been deployed, various threats may also happen.

- Hardware Trojans. HTs inserted during previous phases may stay inactive up till now to evade detection during test time. After an electronic device is deployed, adversaries may choose to active these HTs through executing a specific sequence of instructions, sending a specific data entry, bringing the environment to a specific condition, etc. After activation, HTs will perform their intended attacks (change functionality, leak information, cause DoS, etc.)

on the device.

- Recycling. Adversaries may recycle used products from the market and then refurbish them and resell them for profit. These recycled products not only reduce the profit of the designer, but also pose threats to end customers because these products usually have reliability issues.

- Reverse-engineering. Adversaries may reverse-engineer the device to gain knowledge at various levels (system level, transistor level, etc.). These extracted information may be used to make counterfeit products, or facilitate future attacks.

- Brute-force attacks. In this form of attacks, the adversary only has black-box access to the system (i.e. the adversary can only observe the output of the system with the input chosen at his discretion) and tries to figure out the underlying secret data.

- Side-Channel Attacks (SCAs). Side-channels are defined as the unintended output channels from the physical implementation of an algorithm [9]. These side-channels include execution time, power consumption, electromagnetic radiation, sound, visible light, heat, faulty output, etc. Among these side-channels, execution time, power consumption and faulty output are the mostly exploited ones. In SCAs, the attacker has access to some side-channel information in addition to inputs and outputs. These side-channel information may be correlated with the secret data being processed. Therefore, by observing these

side-channel information, adversaries may learn part of or full secret data.

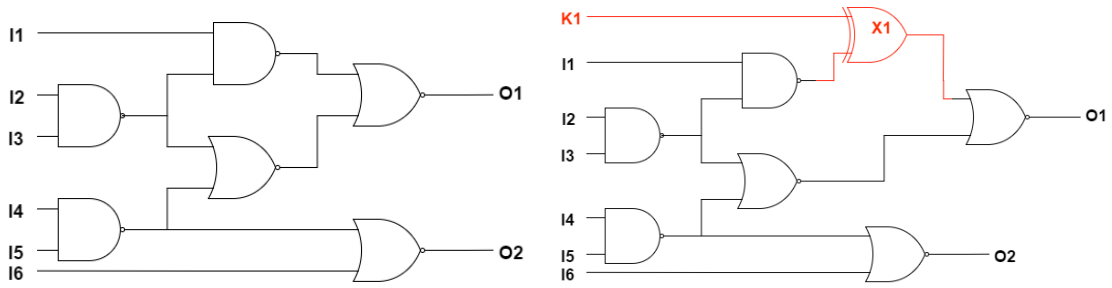
1.3 How Can We Defend

Because there are tremendous number of threats and the potential consequences are devastating, researchers have spent years of effort in enhancing the security of electronic devices. In this section, we will introduce some of the existing countermeasures and their limitations. Note that pure software techniques are beyond the scope of this thesis. We will focus on hardware-assisted countermeasures.

The effort to enhance hardware security typically falls into two categories: design techniques and verification techniques. Design techniques aim to come up with a design that is secure in nature against possible threats. For example, circuit obfuscation techniques can be used to thwart foundries from overproducing. They can also be used to prevent attackers from learning any meaningful knowledge by reverse-engineering. Verification techniques try to verify the reliability, authenticity, and correctness of the underlying system. For example, functional testing, or providing testing vectors to the system and comparing the outputs with intended outputs, is one verification technique. We will explain these two techniques next.

1.3.1 Design Techniques

One way to thwart IP piracy and overproducing is to use logic encryption [10–13]. Logic encryption inserts additional gates into the original design to hide the functionality. One example is shown in Figure 1.1. In addition to primary inputs (I1-



(a) Original circuit (b) Encrypted circuit

Figure 1.1: An example of logic encryption. The circuit is encrypted using one key gate X1. O1 will not outout the correct value unless a correct key K1 is given.

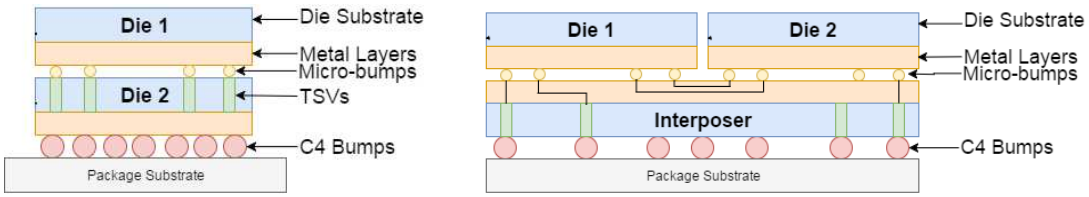
I6), the encrypted circuit takes an additional input (K1) as the key. The circuit will not function correctly (O1 will not output correct value) unless a correct key bit is supplied. The key is given by the designer and applied at the test time. Therefore, the attackers in the fabrication phase cannot use or overproduce any functional copies without the correct key. Moreover, this adds complexity to the structure of the circuit, making it harder for the attacker to tell useful information from the circuit. As a result, it complecates reverse-engineering attacks and hardware Trojan insertions.

Similar ideas are also applied to obfuscate sequential circuit [14, 15]. After the circuit with intended functionality is designer, the designer inserts additional states into the original finite state machine (FSM). These extra states compose the locking structure and unless a correct key is supplied, the FSM will not enter the correct states for normal functionality. After additional states are inserted, the design undergoes the same synthesis, mapping, place and route process and the resulting layout is sent to the foundry for fabrication. Attackers in the foundry

cannot overproduce or steal the design without knowing the correct key.

To further protect the circuit from being reverse-engineered, circuit camouflaging techniques are introduced [16]. The idea is to use a configuratble CMOS cell to disguise the functionality of XOR, NAND or XOR. More specifically, the layout of each configurable CMOS cell is almost the same and only differs in the interconnection. Depending on how the interconnection is made, the CMOS cell can be XOR, NAND or NOR gate. Since it is very hard for the attackers to figure out the interconnection information, the XOR, NAND and XOR looks essentially the same in the layout level. This prevents an attacker from learning the structure of the circuit by reverse-engieering. Consequently, it also prevents an attacker from launching further attacks as introduced in Section 1.2.4.

Split manufacturing has also been proposed to mitigate IP piracy and hardware Trojan insertion during the fabrication process [17–21]. The idea of split manufacturing is to split an IC netlist into multiple parts. Each part is fabricated at a different foundry. As a result, since no single foundry has access to the full design, the IP piracy and hardware Trojan insertion attacks can be prevented. Currently, the most widely studied form of split manufacturing is to split the IC into two parts. One part contains all the active components (transistors) and some wires. This part is referred to as Front-End of Line (FEOL). The other part contains the rest of the wires and is referred to as Back-End of Line (BEOL). Usually, FEOL is sent to untrusted foundry for fabrication while BEOL is fabricated at the trusted or in-house foundry. The reason is that in-house or trusted foundries often have less advanced manufacturing technologies (e.g., 32 nm process) while the untrusted,



(a) Stacked 3D IC (b) Interposer-based 2.5D IC

Figure 1.2: Structure of 3D IC and 2.5D IC.

off-shore foundries have more advanced technologies (e.g., 14nm process). FEOL requires smaller feature sizes and advanced fabrication processes and is better fabricated at untrusted foundries. BEOL on the other hand, does not need advanced processes and can thus be fabricated at in-house foundries. In this way, the untrusted foundry only has access to partial netlist information and cannot steal the full IC design.

Similar to split manufacturing, 3D (and 2.5D) IC fabrication technology also seem to be a promising solution to mitigate IP piracy in the fabrication phase [22,23]. Structure of 3D and 2.5D IC is shown in Figure 1.2. In a 3D IC, separate dies are connected together using through-silicon vias (TSVs) while in a 2.5D IC, connections are made using an interposer. Note that 3D (2.5D) IC is proposed because transistor feature size shrinking is reaching physical limit. 3D IC is deemed as a viable option to continue performance improvement beyond transistor scaling because it allows multiple dies (therefore more transistors) to be stacked up. However, researchers have found out that 3D (2.5D) IC also benefits security. In 3D (2.5D) IC, separate dies can be manufactured in separate foundries. Because no foundries have access to the full netlist, the IP piracy during fabrication is hindered.

To mitigate side-channel attacks, several types of techniques have been introduced [24–27]. The first one is hiding. The goal of randomization-based approaches is to make side-channel information independent of the secret data or operation. To achieve this, the side-channel information is either randomized or kept equal for all secret data or operations. For example, to defend against power side-channel attacks, the designer can design the device such that the same amount of power is consumed in each clock cycle. However, the ideal goal of perfectly random or equal is very hard to be reached in practice. Researchers have been trying to get close to this goal. The second type of techniques is masking. The basic idea is to randomize the secret data that is processed by the device such that the side-channel information is independent of the secret data. In a masked implementation, each secret value v is concealed by a random mask m: $v_m = v * m$. The operation $*$ is usually XOR function $\oplus$, modular addition $+$ or the modular multiplication $\times$. The mask m is generated internally and varies from execution to execution. Since the attacker does not know m, even if he/she figures out the dependency between side-channel information and v_m, he/she cannot learn v.

1.3.2 Verification Techniques

While design techniques try to prevent attacks from happening, adversaries may still be able to evade those defense mechanisms. Therefore, it is important to have verification techniques which will raise red flags when an attack has been detected. In this section, we will introduce some verification techniques to detect

hardware Trojans, IC counterfeiting and recycling. Typically, these verification technique can be divided into four catagories: physical inspection, functional testing, side-channel analysis, and run-time monitoring.

Physical inspections can be used to detect physical and material properties and thus can be used to detect physical counterfeit defects [28–30]. This includes the inspection of the package as well as leads of the chip under test. Scanning acoustic microscopy (SAM) is used to study the structure of a component. It generates an image of the component based on its acoustic impedance at various depth. Any delamination or cracks will be easily detected. After decapsulation, internal structures and wires can be inspected. Scanning electron microscopy (SEM) can also be used to take images of the die, package and leads to detect any anomaly. Chemical composition of the chip under test is also inspected to detect defects related to materials.

Functional testing is used to detect any defects that impact the functionality [31–33]. In functional testing, test vectors are provided to the chip under test. The resulting outputs are recorded and compared with desired ones. Any mismatch between the two indicates a possible defect. With the help of automated test pattern generation, the functional testing process has been highly automated, making it the most efficient way to verify the functionality of a component. However, the biggest challenge in functional testing is that that usually test vectors cannot cover the entire input space. This leaves the possibility that a defect changes the functionality but is undetected. Another form of functional testing is memory tests. In memory tests, read/write operations are performed on a memory to verify its functionality.

Because functions of memories are simple, exhaustive memory tests are possible and are widely used during the process of manufacturing testing.

In side-channel analysis, the side-channel parameters (e.g., delay, power consumption, etc.) of the chip under test are measured. A recycled chip may have a different delay parameter than a brand new legitimate chip. A hardware Trojan-infected chip may consume different amount of power than a Trojan-free chip. Therefore, by measuring the parameters of the chip under test and compare those values with desired values, IC recycling and hardware Trojans can be detected [34–36]. There are two challenges associated with this technique. The first one is how to determine the desired values of all the measured parameters. In case of hardware Trojan detection, this requires a Trojan-free IC (golden IC). However, how to verify or obtain a golden IC remains unknown. The second challenge is that process variation and normal aging may cause the measured parameters to vary from their nominal values. This fact may cause high false alarm rate.

Run-time monitoring serves as the last line of defense to detect defects or malicious behaviors in the hardware system [37–39]. Run-time measurements are compared with simulation data or calculated data. Any anomalous behavior indicate a potential hardware Trojan attack or counterfeit detect. When the anomalous behavior is detected, built-in defense mechanisms can take effect to minimize potential damages. For example, once a hardware Trojan is detected in the communication system, the run-time monitor may choose to use an alternative system to avoid secret data from being leaked. Similar to side-channel analysis, how to determine the desired values of the measured parameters is tough question. In addition, run-time

monitoring often incurs heavy performance or resource overhead, further limiting its usage.

1.4 Contribution and Thesis Organization

In this dissertation, we investigate innovating solutions to mitigate attacks in the hardware systems introduced above. We also explore the implication of emerging technologies and hardware primitives on security. Specifically, we focus on the following three areas:

- Hardware Trojans: Our first part of research focuses on hardware Trojans because they are the most challenging hardware threats to prevent and detect. As introduced above, hardware Trojans can cause devastating consequences. They can also leave backdoors to be exploited by software attacks. Detection and prevention techniques are urgently needed. However, they are no easy tasks. Hardware Trojans can be inserted at any phase during the hardware design cycle. They are also stealthy in nature and are usually small in size, making them very hard to detect. In this dissertation, we propose a novel, efficient and effective Trojan detection approach that eliminates the need for a golden chip. Our approach takes advantage of reverse-engineering and is thus among the strongest approaches. Experimental results on publically available benchmarks show that our proposed method can detect Trojans with more than 99% accuracy and less than 1% false positive rate. We also provide a design-time technique that can increase the sensitivity of the design to hard-

ware Trojans, which can detect 16.87% more Trojans with only 7.87% area overhead and 17.72% leakage power overhead.

- Countermeasures for Timing Side-Channel Attacks on Cache: Our second part of research focuses on mitigating cache timing side-channel attacks. It has been demonstrated that these attacks are able to break many widely-used modern ciphers and are thus very dangerous. Existing countermeasures usually have very heavy overhead, limiting their usage. In this dissertation, we investigate how to defend these attacks with the help of 3D integration technology. We demonstrate that these attacks can be mitigated effectively while still achieving around 10% performance boost over a 2D baseline system.

- Timing Side-Channel Attacks on ORAMs: Our last part of research is focused on the security of a new hardware primitive, i.e. Oblivious RAM (ORAM). It has been demonstrated that memory access patterns can leak very sensitive information even if the underlying data is encrypted. To mitigate this leakage, ORAM has been proposed to conceal the actual access pattern from an adversary who is observing the accesses to the remote storage. In this dissertation, we show that timing side-channel attacks are able to leak sensitive data from many efficient ORAM implementations. We also provide some general guidelines to make ORAMs more secure.

The rest of the dissertation is organized as follows. In Chapter 2, we discuss hardware Trojans, timing side-channel attacks on cache and ORAMs in greater detail. Chapter 3 introduces the motivation and implementation details of our Trojan

detection and design-time technique. These results have been published in [40–42]. In Chapter 4, our countermeasures to mitigate timing side-channel attacks on cache are discussed. Part of the results is published in [43]. Chapter 5 shows our findings on possible timing side-channel attacks on ORAMs. The result will be published in [44]. Finally, Chapter 6 concludes the thesis and discusses future work.

Chapter 2: Background

2.1 Hardware Trojans and Detection Techniques

2.1.1 Hardware Trojans

Hardware Trojans (HTs) are malicious modifications to the circuitry of an IC that can be made at any untrusted or outsourced phase of the IC's production (design, synthesis, fabrication, and distribution) [6]. HTs can leak secret information, disable the IC, or even destroy the IC [7] making them very dangerous. HTs can be described in terms of their physical and activation characteristics [7, 8].

Physical: A HT can change the functionality or parameters of the IC. Functional changes include addition and deletion of transistors, gates, interconnects, etc. Parametric changes consist of thinning interconnects, weakening flip-flops, increasing susceptibility to aging, etc. which can reduce the yield and reliability of an IC design [45].

Activation: Some HTs are always active (eg. parametric Trojans) while others rely on triggering mechanism. A triggered-Trojan consists of two parts: a trigger and payload. The trigger is a sensing circuitry that waits for an event, such as a rare input pattern or internal state, to take place. Before the Trojan is triggered, it is

said to be in the inactive state and the IC mainly works as intended. Once the HT is triggered, the payload gets activated and executes the Trojans attack.

2.1.2 Trojan Detection Techniques

Hardware Trojans (HTs) can seriously degrade the performance and reliability of electronic systems. The consequences of HTs range from loss of profit if inserted in consumer-electronic devices to life-threating if inserted in military devices. As a result, researchers in academia as well as industry have proposed different approaches to detect HTs. These methods fall into the following three categories.

Test-time approaches consist of post-silicon tests and are the most widely studied approaches in the literature. There are two types: functional testing and side-channel fingerprinting [46]. Functional testing [47–49] aims to detect HTs that change the functionality (primary outputs) of the IC from the intended one. Side-channel fingerprinting [50–53] is an alternative approach that measures side channel signals (timing, power, etc.) and uses them to distinguish genuine ICs from Trojan-infested ones. It does not require HT to be triggered to be detected (trigger itself affects the IC's side channels). Most of these test-time methods require a golden IC to compare with. The main disadvantages are that these methods require the Trojan to be activated and they cannot detect small Trojans.

Run-time approaches add circuitry that monitors the behavior/state of a chip after it has been deployed. If deviation from the expected golden behavior is detected, additional circuitry can disable the chip or bypass the malicious logic before the HT

can do any damage. [54] provides a good survey of different run-time approaches. The major disadvantages of these approaches are their high resource overhead and assumption that the run-time circuitry is Trojan-free.

Most test-time and run-time approaches assume that a golden model (intended functionality and behavior) is available to compare with. They suggest reverse-engineering be used to obtain such a golden model [54].

Reverse-engineering based approaches apply the reverse-engineering (RE) process (discussed above) to ICs in order to detect HTs. Basically, the design/netlist uncovered by RE is compared to an intended (golden) netlist. Since these approaches are not only time-consuming, but destructive, they have been pursued the least for HT detection. Their main use of RE has been to verify the Trojan-free chips used in the golden model development [54].

Since all of the above have their own advantages and disadvantages, one proposed direction is to apply each for the highest HT coverage. For example, post-silicon, RE-based approaches can verify golden chips required for test-time and run-time golden models. Functional and side-channel approaches can be used to detect small and large Trojans respectively that were inserted during fabrication. Run-time approaches can act as a last line of defense.

2.2 Timing Side-Channel Attacks on Cache and Countermeasures

In cache-timing side-channel attacks, the attacker measures some timing information related to cache behavior and correlates that with system's underlying

secret to learn it. There are different ways to categorize these attacks. We adopt the method in [55]. Based on how the timing information is obtained, these attacks fall into the following two categories: external interference-based attacks and internal interference-based attacks.

2.2.1 External Interference-based Attacks

The attacks that fall in this categories happen because the attacker and the encryption (victim) process **share** the same cache. In modern multi-core processors, two processes can run on the same core concurrently. In this scenario, they certainly share the same L1 cache belonging to that core. Or in a different scenario, two processes run on two different cores. However, the last level of cache (LLC) is still shared between these cores and therefore these two processes still share some part of cache. Cache sharing results in interference between two processes, e.g., one's cached data being evicted by the other. This is the key feature of the external interference attacks. As a result of the external interference, the attacker will observe some timing variation depending on whether his data is evicted by the victim's data or not.

Prime+Probe Attack. Tromer et. al. proposed this kind of attacks [3]. A typical attack setting is as follows (see also Figure 2.1). The attacker runs his program on the same core with the victim process and they share some levels of cache (e.g., L1 cache). Then the following steps are repeated: 1) Prime: the entire cache is filled with attacker's data. 2) Idle: the attacker lets the victim process run. 3) Probe: the

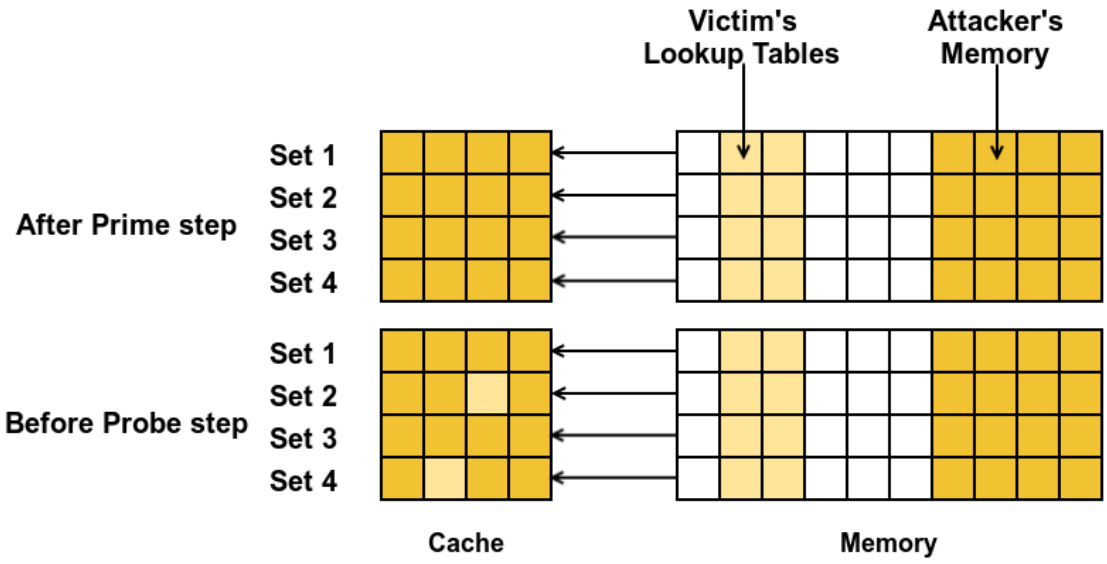


Figure 2.1: How Prime-Probe attack works. Each square represents a cache line. The attacker's data is shown as dark square while the victim's lookup table is shown as light square. The top figure shows the cache state after the prime step. The bottom figure shows the cache state after idle step where two cache lines get evicted by the victim process.

attacker measures the time to reload each set of his data. The reload time of each cache set reflects whether the victim process accesses that cache set. From there, system secret can be leaked.

For example, in [3], the authors correlate the predicate with the highest 4 bits of each byte of the key used in the AES encryption. With some sophisticated statistical methods, the authors were able to recover full 128-bit key used in the AES encryption after only 300 encryptions on Athlon64 and 16000 encryptions on Pentium 4E. In [56], the authors used the predicate learned to attack address space layout randomization (ASLR). ASLR is a mitigation against control-flow hijacking

attack by randomizing the system's virtual memory layout either every time a new code execution starts or every time the system is booted. More specifically, it randomizes the base address of important memory structures such as the code, stack, and heap. As a result, the adversary has no clue about the virtual address of relevant memory locations and thus cannot perform a control-flow hijacking attack. The authors learn the ASLR scheme using the following way. They make sure that a privileged code portion (such as the operating system's system call handler) is present within the cache by executing a system call. Then a designated set of user space addresses are accessed and the system call is executed again. The time to perform the system call is recorded. If the time is longer, then those accesses evict the system call handler code from the cache. Because the structure of the cache is known, parts of the virtual address of the system call handler code is revealed. By repeating this, the ASLR scheme is eventually fully revealed. In [4], the authors show that they are able to learn the specific keys pressed on the keyboard both for GTK-based Linux user interfaces and Windows user interfaces within 1ms.

Note that the same scheme can also be applied to other types of cache. For example, in [57, 58], authors apply Prime+Probe technique to instruction cache. The underlying idea is similar. To learn if a specific instruction cache set has been visited or not, the adversary first fills that cache set with his own instructions by executing instructions that map to that cache set. Then the adversary lets the victim process run. In the Probe step, the adversary re-executes all instructions executed in the Prime step. If the re-execution takes longer, the adversary knows that that particular instruction cache set has been accessed by the victim process. In [57], one

attack on the RSA encryption is shown. The authors were able to identify different RSA phases as well as the operation sequence of sliding window exponentiation used in RSA encryption. Around 200 scattered bits of 512-bit exponents can be leaked.

When Prime+Probe method is applied to the branch target buffer (BTB), we call it Branch Prediction Attack (BPA). Although BTB is not a cache, we categorize this kind of attacks as cache-based timing side-channel attacks because in modern processors, BTB functions similarly to a set-associative cache storing the target address of some previously taken conditional branches. When CPU is executing a conditional branch, it first checks if the BTB has a match. If not, the branch is predicted as not taken. CPU updates the BTB later If the branch turns out to be taken indeed. The Prime step in this attack is to execute some conditional branches that are all taken and are mapped to a given BTB set. The Probe step is to re-execute these conditional branches. Depending on whether the Probe step is longer or shorter, the adversary can learn whether there exist branch mispredictions due to BTB misses and learning that reveals whether some conditional branches in the victim process that are mapped to the same BTB set were taken or not. [5,59] shows that the adversary can successfully break the RSA encryption algorithm using BPA. In [60], authors show how to break the ASLR using BPA. The results show that they can recover the kernel ASLR in about 60 ms when performed on Linux system.

Flush-Reload Attack In [1], the authors outline another method of attack called Flush+Reload attack. The attacker shares the last-level of cache with the victim. The attacker is interested in whether the victim accesses a specific memory address. To learn this, he repeatedly performs the following steps: 1) Flush: the

monitored memory address is evicted from the entire cache hierarchy, usually using some architecture-dependent instruction. For example, clflush instruction on X86 architecture. 2) Idle: the attacker waits while the victim process is running and using the cache. 3) Reload: the attacker reloads the monitored memory address. If during the idle time, the victim has accessed the monitored line, then the monitored line will be cached and consequently the reload time will be shorter than the case where the victim has not accessed the monitored line.

Various information can be leaked subsequently. In [61], the adversaries can learn the full 16-byte key used in an AES encryption after only 100 encryptions. Similar results are obtained in [62], but note that the attack is also successful in a cross-VM environment. In [63], the authors can recover secret keys of the secp256k1 curve, used in the Bitcoin protocol after observing only 25 signatures. Other successful attacks include [64–68].

2.2.2 Internal Interference-based Attacks

In this kinds of attacks, the attacker will rely on the reuse of the cached data within the victim process. Bernstein [2] initiated the work in this area. The basic idea is to exploit the impact of the reused cached data on the total execution time of the victim process. The entire execution time t can be affected by both inputs and underlying secret based on some internal cache contentions. By correlating the execution time with the underlying secret, the attacker can learn partial or full secret data. The attack usually requires large amount of samples to be collected.

Figure 2.2: An illustration of a cache line in Partition Lock cache.

In [2], 2^{27} encryption of random packets are needed to learn the full key used in AES encryption. Similar attacks have been also proposed in [69, 70]

Note that there are other methods to categorize these attacks. For example, Prime+Probe attack and its variant are often called *access-driven attacks*. Bernstein's attack is often categorized as *time-driven attack*.

2.2.3 Hardware Countermeasures

Generally hardware-based mitigation approaches fall into three categories: partition-based, randomization-based and timing source obscuring-based.

Partition-based approaches partition the cache into several parts, with each part used by one process. For example, in [71], the concept of partitioned cache is introduced. The author proposed to partition cache in such a way that no processes share the same cache block. However, no implementation details were given. Such a static partition usually does work well because it incurs heavy performance overhead. In [72], the authors proposed to dynamically reserve cache lines for active threads and prevents other co-executing threads from evicting reserved lines. The performance overhead for 2-threaded workloads are around 1%. In [73], the authors proposed to add instructions to ISA so that secure processes can lock a cache line for its own uses. The modified cache line is shown in Figure 2.2. Each cache line

contains one bit indicating whether the cache line is locked. It also includes ID field which indicates the ID of the process that locks the cache line. If a cache line is locked by a process, only that process can unlock it and no other processes can ever use this cache line. The worst-case performance overhead was reported to be 20%. Similar ideas were introduced in [74]. In [75], the authors proposed to flush the cache on a context switch, which isolates data shared between two processes. However, it incurs extra overhead of flushing the cache. Usually partition-based approaches defend external interference attacks because the attacker's data cannot be evicted by the victim process. However, these countermeasures are ineffective against internal interference attacks and often incur heavy performance overhead.

Randomization-based approaches randomize the side-channel information with the hope that the adversary cannot learn anything useful from the measurements. In [76], the authors proposed to load a random nearby memory block to the cache rather than the one on-demand during a cache-miss. In this way, the correlation between access time and cache hits is reduced. In [77, 78], authors proposed to replace a cache line in a random cache set when contention between victim and the attacker happens. However, the above two methods are still vulnerable to internal interference attacks.

Timing source obscuring-based approaches target at the timekeeping. [79] proposed to add some random delay to the output of the RDSTC instruction, which is used by most, if not all, cache-based timing SCAs. This adds noise to the attacker's measurement, making the attacker measurement much harder. In [80], the authors proposed a system that can defend against timing-based side-channel attacks in

a cloud setting. The system triplicates each cloud-resident guest VM and places replicas so that the three replicas of a guest VM are no overlapped with those of others. Then it uses the timing of I/O events at a VM's replicas collectively to determine the timings observed by each one or by an external observer. In this way, the observed timing behaviors are similarly likely in the absence of any other individual VM and are thus obscured. The major flaw of these approaches is that they do not defend against an attacker who uses his own timekeeping sources.

2.3 Oblivious RAM

2.3.1 General Idea

It has been demonstrated that memory access patterns can leak very sensitive information even if the underlying data is encrypted. For example, in [81], an attack that exploits data access pattern made to an encrypted email repository is proposed. The results show that with very little prior knowledge, the attack can disclose as much as 80% of the search queries. To mitigate these attacks, oblivious RAM (ORAM) has been proposed to conceal the actual access pattern from an adversary who is observing memory traffic.

ORAM was first proposed by Goldreich et. al. in [82] to conceal the following from adversaries observing memory traffics: (1) the locations of the items accessed, (2) the type of the access (either read or write), (3) the relative order of all the data accesses, and (4) how many requests are made to the same location. In their original implementation, ORAM hides the access pattern through continuous shuf-

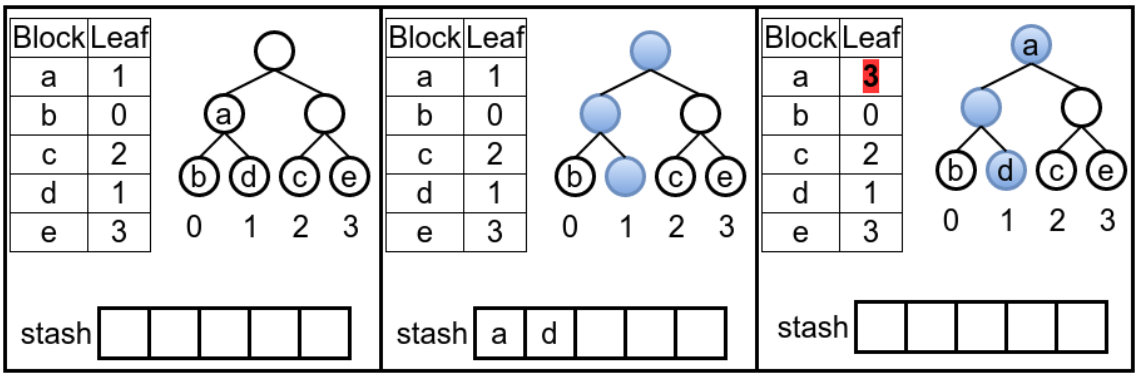


Figure 2.3: An illustration of reading block "a". In this ORAM, $Z = 1$ and $L = 2$.

fling, decryption and encryption operations and has heavy performance overhead. Substantial progress has been made to make ORAM more efficient [83–87].

2.3.2 Path ORAM

The Path ORAM is proposed in [88] as a simple and efficient ORAM protocol. In Path-ORAM, the memory is organized as a binary tree containing $L + 1$ levels (with level 0 being the root and level L being the leaves) and 2^L leaves. Each node in the tree is called a bucket. Each bucket can hold Z data blocks. A data block is analogous to a cache line, which is a fixed number of bytes of data. If a bucket has less than Z real blocks, it is padded with dummy blocks to be size Z. Z is a user-defined parameter and usually a small number suffices, such as $Z = 4$. Let $i \in \{0, 1, \ldots, 2^L - 1\}$ denote the i^{th} leaf. Each leaf i will define a unique path from the root to leaf i, denoted by $P(i)$. Let $P(i, l)$ denote the l^{th} node on the path from root to leaf i.

The client maintains two data structures locally. The *stash* is used to hold blocks that overflow from the tree bucket on the remote storage. The *position map*

is used to store the mapping between a block and a leaf node in the tree in the remote storage. For example, **positionMap**$[b]= i$ means that block b is mapped to leaf i. The main invariant of Path ORAM is: at any time, a block is mapped to a leaf bucket in the tree in the remote storage, uniformly random. That block is either in the stash or in some bucket in the path from the root to the mapped leaf. We show next how this invariant is maintained.

During initialization, the client's stash is emptied. Every bucket in the remote storage is initialized to contain encryptions of some dummy blocks. The position map is filled with independent random numbers between 0 and $2^L - 1$. Whenever an access *op* (either read or write) of a block b is made, the client first looks up the position map to get the leaf that block b is mapped to, denoted by i (i.e.**positionMap**$[b]=i$). Then according to the invariant, if block b exists on the remote storage, it must be either in the stash or on the path $P(i)$. Then each block in the path $P(i)$ is read into the stash and decrypted. At this point, if block b exists on the remote storage, it must be in the stash. Then if $op = read$, block b is returned, otherwise, it is updated. Next, block b is remapped to another leaf and the position map is updated. As many blocks as possible in the stash are re-encrypted, and written back to path $P(i)$ in the order of leaf to root. A block b' can be placed at $P(i, l)$ only if the path $P($**positionMap**$[b'])$ intersects the path $P(i)$ at level l. This ensures that invariant is maintained. If a bucket in the path $P(i)$ has less than Z real blocks, it is padded with encrypted dummy blocks to be size Z. One example of reading a data block in the Path ORAM is shown in Figure 2.3.

Chapter 3: Reverse-engineering Based Hardware Trojan Detection

3.1 Introduction

More and more integrated circuit (IC) designers are outsourcing fabrication to foundries and incorporating third-party intellectual property (IP) cores into their designs. This practice helps to lower costs and reduce time-to-market pressure, but can also lead to security problems such as malicious modifications to ICs, also known as hardware Trojans (HTs). HTs can change IC functionality, reduce IC reliability, leak valuable information from the IC, and even cause denial of service [8]. Depending on the applications, the consequences of HTs range from loss of profit if used in consumer-electronic devices to life-threatening if used in military devices.

To reduce the risks associated with Trojans, researchers have proposed different approaches to detect them. Test-time detection approaches [46–49] have drawn the greatest amount of attention from researchers. In these approaches, functional and/or side-channel behavior of suspect ICs are compared to a "golden model" that represents the expected behavior of a Trojan-free IC. If the suspect IC deviates sufficiently from the golden model, it is classified as being Trojan-infected. While these approaches have met with some success, obtaining such golden models/data is mostly an open problem.

Motivation. Prior works such as [54] suggest that reverse-engineering (RE) be used to identify Trojan-free ICs and verify the data used for the golden models. However, our literature survey did not find any work describing how to do this accurately and efficiently. Reverse-engineering is actually a very complex, error-prone, and time-consuming process. It consists of 5 steps [89]: decapsulation, delayering, imaging, annotation, and schematic creation. The first 3 steps essentially obtain images of the physical layout for test ICs. The last 2 steps extract netlists for the circuit/design based on the images. A naive approach to RE-based Trojan detection would apply all 5 RE steps to extract netlists for comparison with a golden netlist, but this approach is flawed. First, some Trojans (eg. parametric Trojans [45]) can actually be missed by only comparing netlists. Second, the effort required by this naive approach is unnecessarily excessive because generating the netlist via the last 2 RE steps is time-consuming and requires manual input.

Contributions. In this Chapter, we propose a more efficient and robust RE approach for solving the above problem. In our approach, we avoid extracting netlists altogether. Instead, we develop a machine learning approach that classifies ICs as Trojan-free and Trojan-inserted based on features extracted from the IC images. Our approach essentially eliminates the last two reverse-engineering (RE) steps 4-5, which can save lots of unnecessary effort. The major features of our approach and our contributions are as follows:

- To our knowledge, we propose the first IC classification scheme for hardware Trojan (HT) detection based on reverse-engineering of chips without generat-

ing a gate or transistor netlist. As stated above, this saves much effort.

- In our approach, we use images obtained from the imaging step of reverse-engineering and extract features from them to characterize an IC's physical layout. We develop a Support Vectors Machine (SVM) approach that automatically learns how to distinguish between expected and suspicious structures in the ICs and ultimately classifies unknown ICs as Trojan-free or Trojan-infected. Our method does not rely on a Trojan-free (golden) IC which is the underlying assumption of many other approaches.

- Our approach depends heavily on several SVM modeling parameters as well as algorithm parameters related to feature selection. We discuss how to select these parameters in a way that makes our classification approach accurate and robust to noise in the training data. Furthermore, our proposed features account for fabrication and reverse-engineering induced variations that occur within the ICs.

- We perform simulation experiments on 6 publicly available benchmarks, ranging in size from 50 to over 100,000 gates. The results show that our method can detect three different kinds of HTs (inserted transistors, deleted transistors, and parametric changes) with very high accuracy. We also vary the modeling and algorithm parameters to determine their impact on our method.

The rest of the chapter is organized as follows: Section 3.2 gives a brief review of reverse-engineering, hardware Trojan characteristics and detection, and general SVM. Section 3.3 defines the hardware Trojan detection problem we want to solve

and discusses the motivation behind it. Section 3.4 explains our SVM-based Trojan detection method in detail including challenges, feature selection, SVM implementation, parameter selection, and merits. In Section 3.5, experimental results are discussed. In Section 3.6, we discuss the motivation behind the design-for-security technique. Section 3.7 explains the design-for-security technique in greater detail. Relevant experimental setup and results are shown in Section 3.8. Finally, Section 5.5 concludes the paper.

3.2 Preliminary

3.2.1 Reverse Engineering

Reverse-engineering of an IC is the process of analyzing an IC's internal structures, connections, etc. in order to determine how it was designed and how it operates. RE is considered an important tool for learning how to build and improve upon designs as well as proving Intellectual Property (IP) infringement. A typical RE flow includes the following steps [89].

1. *Decapsulation*: The die is removed from its package.

2. *Delayering*: Each layer of the die is stripped off one at a time using chemical methods while polishing the surface to keep it planar.

3. *Imaging*: Thousands of high-resolution images of each exposed layer are taken using scanning electron microscope (SEM). The images are stitched together to form a complete view of the layer using special software. Multiple layers

are also aligned at this step so that contacts and vias are lined up with layers above and below them.

4. *Annotation*: All structures in the device (interconnects, vias, transistors, etc.) are annotated either manually or by using image recognition software [89].

5. *Schematic creation, organization, and analysis*: A hierarchical or flat netlist is generated using the annotated images as well as public information (datasheets, papers, etc.).

All of the above steps are time-consuming and error-prone. The first 3 steps essentially extract images of the structures contained in the IC. Removing and planarizing layers affect the structures in the exposed and lower layers, resulting in additional noise in their IC structures. The last 2 steps are required for circuit/design extraction and are also quite challenging. The annotation process and the schematic creation/analysis often requires input from experienced analysts [89].

3.2.2 A Survey of Design-for-Trust Strategies

Design-time strategies aid test-time detection approaches [90]. They aim at either preventing the insertion of Trojans or easier detection of Trojans. Very few works have been proposed in this direction and we provide a survey of design-time strategies in this section.

In [91], a new systematic design for Trojan test strategy is proposed. It identifies in the RTL code Trojan-vulnerable source code and replaces it with hardened Trojan prevention and detection code. It also embeds some probe cells into the

design. The goal is to make Trojan insertion harder.

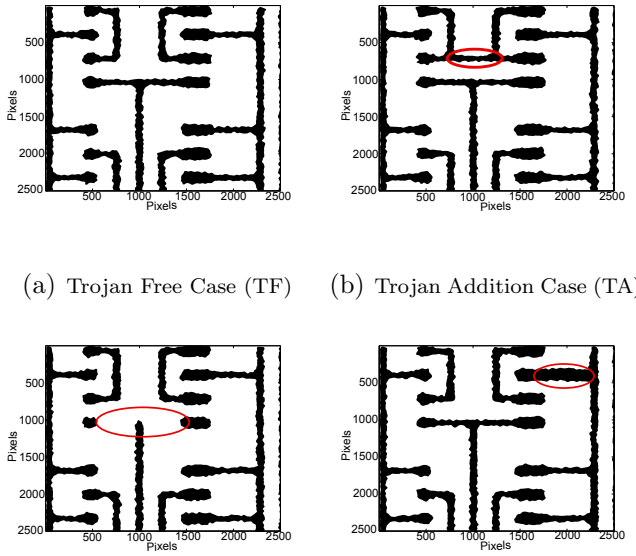
In [92], an inverted voltage scheme is proposed to pronounce the behavior of any undesirable logic. The authors claim this technique will better activate and detect the malicious insertions in third party ICs. In [93], an efficient dummy flip-flop insertion procedure is proposed to increase Trojan activity. To increase the probability of Trojan activation during test-time, [94] proposes the insertion of scan flip-flops. [95] maximizes circuit activity in specific regions of the IC while reducing that in other regions to enhance side-channel analysis

3.3 Problem Statement and Motivation

Problem Statement. Assume we are given ICs from one or more untrusted foundries. We want to determine which ICs are Trojan-free (TF) and which have Trojans inserted (TI). We consider that there are three-types of TI cases possible:

- *Trojan Addition (TA)*: These HTs add transistors, gates, and interconnects into the original layout.
- *Trojan Deletion (TD)*: These HTs delete transistors, gates, and interconnects from the original layout.
- *Trojan Parametric (TP)*: These HTs perform physical changes to the transistors, gates, and interconnects of the original layout as suggested in [45].

Examples of TF, TA, TD, and TP are shown in Figure 3.1. Note that the above problem can be viewed as an instance of the classification problem which is given



(a) Trojan Free Case (TF) (b) Trojan Addition Case (TA)

(c) Trojan Deletion Case (TD) (d) Trojan Parametric Case (TP)

Figure 3.1: Example of three kinds of trojans. SEM image of metal1 layer is shown.

as follows. Assume we have 2 classes of objects denoted by C_0 and C_1. Let each object be represented by a feature vector $\mathbf{x} = (x_1, \ldots, x_n)$ where x_i denotes the ith feature, $x_i \in \mathbb{R}$, and n denotes the # of features. Given an unknown object A, the problem is to determine the correct class of A. In our case, objects are chips under test and TF represents class C_0 while the TI cases (TA, TD and TP) represent class C_1.

Motivation. As discussed in Section 2.1.2, prior works [54] suggest reverse-engineering (RE)-based methods be used to identify Trojan-free ICs in order to develop the golden models for other HT detection approaches. However, there has not been any work devoted towards performing this classification. One can only assume that papers in the literature would assume a golden netlist and naively apply all 5 RE

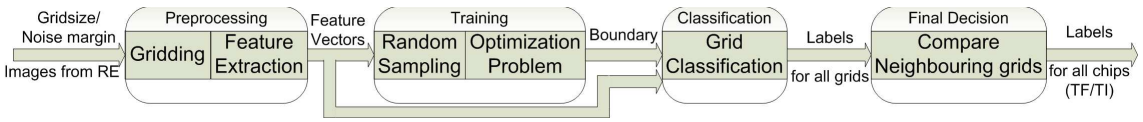


Figure 3.2: Block diagram of our Trojan detection approach.

steps (see Section 3.2.1) to extract netlists from all unknown chips. Only data from those chips with netlists that match the golden netlist would be included in the golden model.

We feel that there are some issues with this naive approach. First, some Trojans (eg. parametric Trojans) can actually be missed by only comparing netlists. Second, the effort required by this naive approach is excessive. RE steps 4-5 are time-consuming and require manual input for annotation and schematic read-back. In this paper, our goal is to develop a more efficient and robust RE approach for solving the above classification problem. In our approach, we avoid extracting netlists altogether. Instead, we develop a one-class SVM-based approach that makes a classification decision based on features extracted from the IC images. The key features of our approach are ability to detect all the above Trojan cases in an *efficient and automated* fashion. Details of our approach and challenges we overcome are discussed in the next section.

3.4 SVM-based Trojan Detection

Assumptions. As discussed above, we wish to solve the following classification problem: Given N ICs, classify each as Trojan-free (TF) or Trojan-inserted (TI). To perform the classification, we assume the following. We assume that the original de-

sign is Trojan-free and Trojans have only been inserted during fabrication. Formal verification and trusted (in-house) design are two approaches that guarantee this. We also assume that we have the original physical layout (referred to as the *golden layout*) for all the layers of the IC. Since the designer would ultimately come up with this layout and send it to the foundry, it is reasonable to assume we have this.

Basic idea. Our approach utilizes the first 3 steps of reverse-engineering (Decapsulation, Delayering, and Imaging) to obtain internal images of each layer (poly, metal1, etc.) for the ICs. By omitting reverse-engineering (RE) steps 4-5, we save lots of unnecessary effort. The images recovered represent the physical structures and layout of the ICs. Then we classify the ICs using these images and support vector machines (see below). In contrast to the naive RE approach which requires manual effort, our approach is fully automated and more efficient in terms of computational and storage resources.

Classification via SVM. The key of our approach is to solve a classification problem which is defined in Section 3.3. There are many machine learning approaches to perform classification. Our approach will make use of the popular Support Vector Machine (SVM) [96] approach. Classification via SVM usually involves two phases:

- *Training:* In this phase, SVM takes as input training dataset $DS_T = \{(\mathbf{x}_j, y_j) \mid j = 1, \ldots, |DS_T|\}$ where $\mathbf{x}_j$ and y_j are the jth feature vector and its class label (C_0 or C_1). An optimization problem that relies on the dot product of feature vectors is solved to find a linear decision boundary $\boldsymbol{\omega}$ which separates the feature vectors of class C_0 from class C_1 so that as many objects from DS_T as

possible are correctly classified.

- *Classification:* This phase takes as input a feature vector $\mathbf{x}^*$ of an unknown object and outputs its predicted class based on which side of the decision boundary $\boldsymbol{\omega}$ that $\mathbf{x}^*$ resides.

Challenges. The main component of our approach is SVM. Applying standard SVM discussed above to our problem has its own set of unique challenges:

1. *SVM features.* Selecting relevant features for SVM-based classification is important for accuracy and computational efficiency. In our case, we need to extract features (see feature vectors defined in Section 3.3) from the IC images obtained from step 3 of the RE process. These feature vectors would be used during the classification process. Choosing too many features will result in large overheads. Also, the features selected must contain enough relevant information to accurately classify the ICs as Trojan-free (TF) or Trojan-inserted (TI). In our approach, we use a heuristic for selecting features. Features are determined by comparing physical layouts in the RE images with the golden layout. Such features are appropriate because they give us some estimate for how much deviation from the golden layout exists in the ICs. More details on the features are discussed in Section 3.4.1.

2. *Fabrication and RE-induced variations.* Differences between the ICs are bound to exist because of random fabrication variations. Furthermore, the RE process (delayering) is also imperfect and could result in additional noise in the IC structures. A naive matching of the designed layout with the RE layout

would always result in a mismatch due to the existence of variations. The challenge is to detect the Trojans while accounting for the variations caused by manufacturing and RE process. Our SVM classifier must be able to distinguish between these random differences and the systematic differences caused by Trojan insertion. In our approach, we estimate how much random noise is to be expected between the RE layout and the golden layout. Noise margins are introduced when extracting feature vectors. More details are given in Section 3.4.1.

3. *SVM Training.* Traditional two-class SVM problem requires training samples from both classes to correctly train the classifier. However, in our HT detection problem (as defined in Section 3.3), we cannot know in advance which ICs are Trojan-free and which are Trojan-inserted, nor do we know what kind of modifications the attackers will make to those ICs. This leads to a lack of traning samples from one class. We overcome this by using a one-class SVM approach. More details are given in Section 3.4.2.

4. *Computational Effort.* Another general issue that we encounter is that extracting features, training the SVM, and performing classification *for the entire layout* can be demanding in terms of computational effort and storage. Rather, in our approach we simplify each of these steps by breaking them up into smaller sub-problems. Specifically, we divide the IC images into smaller non-overlapping grids (see Figure 3.3). Then we independently extract features and classify each grid as Trojan-free (TF) or Trojan-inserted (TI) separately.

This griddingapproach is advantageous because it allows us to parallelize or distribute the processing for each grid. A final classification decision for the entire IC (TF or TI) is made by examining the classes determined for its grids.

Overall Algorithm. Our overall approach is summarized in the block diagram shown in Figure 3.2. We take as input the golden layout, N chips to classify, and parameter values for grid size, noise margin d_{nm}, etc. The N chips undergo only first 3 RE steps, which result in images of each layer for all N chips. The next step is to divide the images for each layer of all N chips into non-overlapping grids. Features are extracted for all N chips for each grid in each layer. We train the classifier and obtain a decision boundary for each layer using a subset of the chips (*possibly even just one chip*). After training, we classify the grids in the each layer of all the N chips as Trojan-free (TF) or Trojan-inserted (TI) based on the ν-SVM decision boundaries of each layer. Finally, we determine a label for each chip based on these grid classifications.

3.4.1 Feature Selection

As discussed above, we break the images (layouts) into smaller non-overlapping grids as shown in Figure 3.3. Features are extracted from each grid by comparing the corresponding grids of the IC under test (ICUT) with the golden layouts grids (which we know from design). Note it is assumed that step 3 of the RE process (see Section 3.2.1) has already properly aligned the images. We will experiment with different grid sizes in Section 3.5.

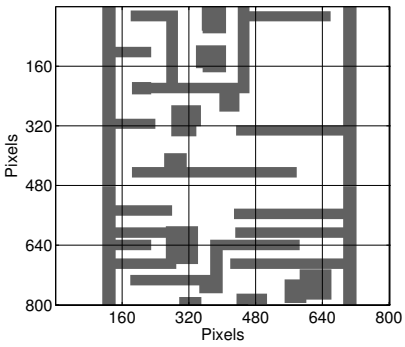


Figure 3.3: Gridding techniques. Part of metal1 layer of s298 benchmark with gridsize 160×160 pixels is shown.

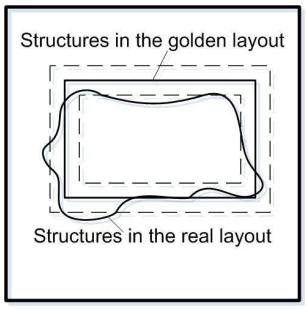


Figure 3.4: Golden layout and SEM image of real layout within one grid. Solid rectangle and curve denote Z and Y respectively while rectangles in dashed line are Z_{in} and Z_{out}.

We illustrate many of our features with the help of Figure 3.4. The RE image and golden layout of one grid are shown. Let Y denote the structures in the layout image obtained by reverse-engineering, which is the curly shape in the figure. Let Z denote the structures in the golden layout, which is the solid rectangle in the middle. Let Z_{in} be Z scaled by a margin d_{nm}. Let Z_{out} be Z expanded by a margin d_{nm}. Z_{in} and Z_{out} correspond to two rectangles in dashed lines. d_{nm} represents noise in fabrication and reverse-engineering process that is thought to be reasonable. Let $\overline{Z}$ denote the complement of Z, where the universal set is the set containing all the pixels in the grid (hence $\overline{Z}$ is the set of pixels outside Z).

We select five features which are determined based on area and centroid differences between the RE and golden layouts.

Area differences. Our first 3 features are obtained by calculating the intersection of different areas between the golden layout and reverse-engineered layout. They are

given by the following equations

$$f_1 = \frac{A(Y \cap Z_{in})}{A(Z_{in})} \tag{3.1}$$

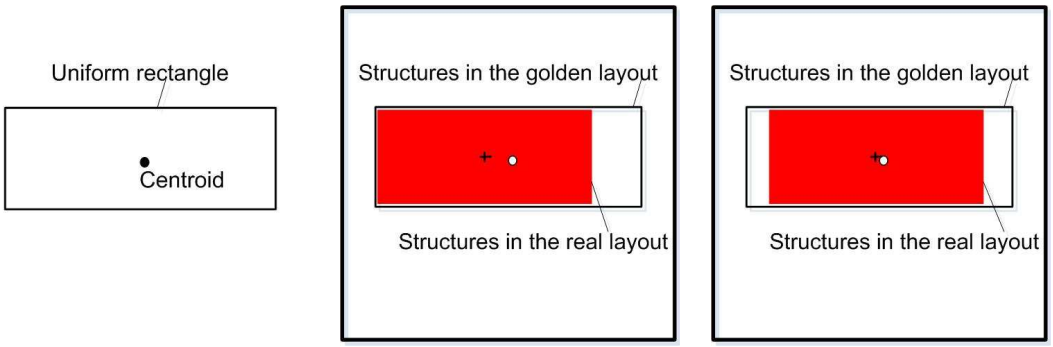
$$f_2 = 1 - \frac{A(Y \cap \overline{Z_{out}})}{A(Y)} \tag{3.2}$$

$$f_3 = 1 - \frac{A(Y \cap \overline{Z_{out}})}{A(\overline{Z_{out}})} \tag{3.3}$$

where $A(Z)$ denotes the area of Z. Equation (3.1) captures how much from the golden layout is missing in the reverse-engineered layout while Equation (3.2) and (3.3) concentrate on whether there is anything additional in the reverse-engineered layout compared to the golden layout. The values of these features should be 1 if no modifications exist. When $A(Z_{in}) = 0$, which means there is no structure at all in the grid, we set $f_1 = 1$ meaning there is no deletion because there is nothing to delete. When $A(Y) = 0$ or $A(Z_{out}) = 0$, which means either there is no addition or there is no way to add because a grid of golden layout is entirely filled with structure, we set $f_2 = 0$ or $f_3 = 0$ respectively.

The reason that we expand and shrink original structure by a margin and use them to get first three features is based on the observation that process variations and noises in RE process tend to produce imperfections only outside Z_{in} and inside Z_{out}. Thus, any differences within these areas are regarded as acceptable while differences outside these areas are suspicious. The value of d_{nm} can be reasonably determined by performing simulations of the fabrication and RE processes. In Section 3.5.2, we shall vary this parameter to see its effects.

Centroid differences. A centroid is defined as the geometric center of a mass (eg. see Figure 3.5). We calculate features based on the difference between the centroid of



(a) (b) (c)

Figure 3.5: Illustration of centroid and use of centroid difference to distinguish two cases. The cross and open dot denote the centroid of structures in the golden layout and real layout respectively.

the golden layout and ICUT layout because not only do area differences matter, but where these differences occur also matters. Figure 3.5 explains this in detail. The solid rectangle in the middle is the structure in the golden layout while the shaded area is the structure in the RE image for the ICUT. In both cases, $A(Y)$ is the same. But in Figure 3.5(b), Y is more biased to the left which indicates malicious deletion of the right portion. In Figure 3.5(c), Y is more towards the center which indicates random reverse-engineering and fabrication noise. Both cases will produce the same f_1, f_2, f_3, but centroids will capture the difference.

Equations for the centroid difference features are as follows

$$f_4 = \frac{|CX(Z) - CX(Y)|}{grid's\ length} \tag{3.4}$$

$$f_5 = \frac{|CY(Z) - CY(Y)|}{grid's\ height} \tag{3.5}$$

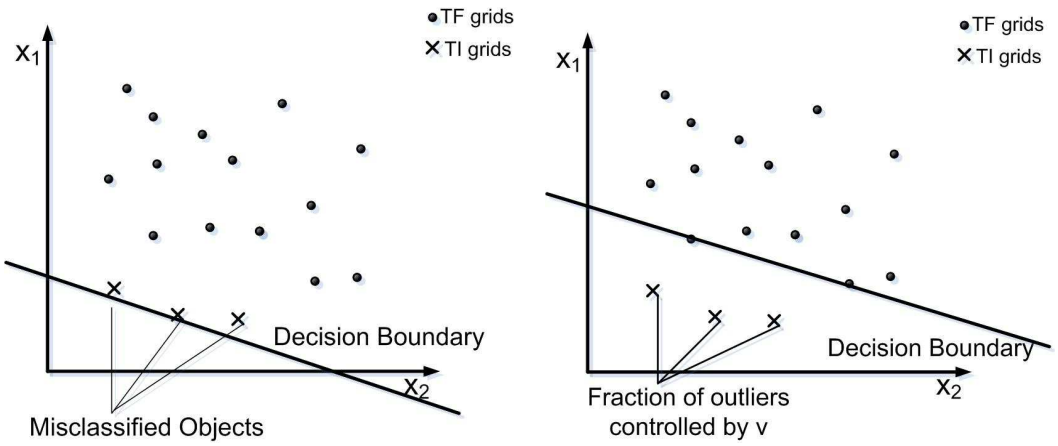
where $CX(Z)$ and $CY(Z)$ denote the x and y coordinate of Z's centroid.

3.4.2 SVM implementation

One issue that arises is that we do not have labels for training samples for either class in our HT detection problem because we do not know which IC grids are Trojan-infested, nor do we know what kind of modifications are made. To overcome this, we make a simplifying assumption and use a special implementation called one-class ν-SVM [97].

The assumption is as follows: we assume that most of the IC grids are samples of the Trojan-free (TF) class. This is reasonable because Trojans should only infect a small portion of each IC. Otherwise, if the Trojan was large, it would be easily detectable by other HT detection methods (eg. functional and side-channel analysis). Then one-class SVM will modify the optimization problem to find a decision boundary $\boldsymbol{\omega}$ that closely surrounds those training samples. Given a new sample, it will then be classified as Trojan-free or Trojan-infested depending on which side of the boundary it lies in. [98] proposes different decision boundaries like hyperplane, hypersphere, and hyperellipsoid. In our Trojan detection context, we will use hyperplane because it has a shorter run time and its performance is comparable to the other decision boundaries according to our experimental results.

One issue with this assumption/approach is that a small portion of the grids/samples may actually be Trojan-infested. Hence the boundary chosen by one-class SVM will include these samples (outliers) which will result in misclassification (see Figure 3.6(a)). ν-SVM introduces a parameter ν to limit the number of outlying samples (in our case TI samples) that are contained within the boundary. This makes our



(a) Without ν controlling fraction of outliers (b) With ν controlling fraction of outliers

Figure 3.6: Parameter ν will control fraction of outliers. Those outliers in the training samples will not be included in the decision boundary with proper choice of ν.

approach more robust to imperfections in the training set labels and should increase the classification accuracy (see Figure 3.6(b)).

The main details and formulation of the ν-SVM approach are discussed below.

ν-SVM Training and Classification. [97] gives a detailed algorithm of one-class ν-SVM. It introduces a parameter $\nu \in (0, 1)$ which bounds the chance of accepting outlier sample points. We summarize it below in the context of our IC classification problem.

The first step is training. We assume we have a training dataset $DS_T = \{(\mathbf{x}_j, y_j) \mid j = 1, \ldots, m\}$ where $\mathbf{x}_j$ and y_j are the jth grid's feature vector and its class label (TF). We determine a hyperplane (boundary) whose normal vector is $\boldsymbol{\omega}$ to separate as many training samples (grids) as possible from the origin with a maximum margin ρ. Errors (representing samples lying on the same side of the

hyperplane as the origin does) are allowed and are denoted by slack variables ξ_i. To find such a hyperplane, one needs to solve the following quadratic programming problem:

$$\min_{\boldsymbol{\omega} \in F,\, \rho \in \mathbb{R},\, \boldsymbol{\xi} \in \mathbb{R}^m} \quad \frac{1}{2}\|\boldsymbol{\omega}\|^2 + \frac{1}{\nu m}\sum_{i=1}^{n} \xi_i - \rho \tag{3.6}$$

subject to

$$\boldsymbol{\omega} \cdot \mathbf{x}_i \geq \rho - \xi_i, \quad \xi_i \geq 0, \quad i = 1, 2, \ldots, m$$

In the above, $\boldsymbol{\omega}$ and ρ are the normal and bias of the hyperplane respectively; F is the input feature space; ξ_i are slack variables; m is the number of training samples; ν is a modeling parameter. In the above optimization problem, the ν parameter acts as a weighting term for the slack variables. Smaller (larger) ν results in less(more) training samples being on the same side of hyperplane as the origin is. Equation (3.6) is solved for $\boldsymbol{\omega}$ and ρ which define the hyperplane.

The next step is classification. Let $\mathbf{x}^*$ denote a feature vector of an unknown grid that we wish to classify as TF or TI. $f(\mathbf{x}^*)$ is a function that outputs a label for $\mathbf{x}^*$ given by

$$f(\mathbf{x}^*) = \text{sgn}(\boldsymbol{\omega} \cdot \mathbf{x}^* - \rho) \tag{3.7}$$

Intuitively, the output of f is based on which side of the hyperplane $\mathbf{x}^*$ falls on in the input feature space F. $\text{sgn}(x)$ is the sign function which is 1 for all $x > 0$ and -1 elsewhere. In our case, an output of 1 and -1 corresponds to the TF and TI class labels respectively.

Parameter values. There are several parameters that we need to decide before

solving the above optimization problem:

- ν *parameter:* [97] proves that ν is the upper bound on the fraction of outliers. This means if we know the fraction of grids that are maliciously modified, we can set the value of ν appropriately but the problem is that the fraction is not known beforehand. If we choose ν to be fairly small, say 1e-8, some outliers may not be discovered, causing a high false positive rate (failure to detect HTs). On the other hand, if we choose ν to be a large number, say 0.5, some Trojan-free (TF) grids may be mislabeled as Trojan-inserted (TI), resulting in a high false negative rate (false alarm of HTs). Our observation is that Trojans are more likely to modify polysilicon layer and interconnection layers while only a small portion of via layers may be modified. Thus, we can choose relatively large ν for polysilicon and interconnect layers while keeping ν small for via layers.

  Also, ν should be varied with the number of training samples/grids. For example, if ν is so small that $\nu \times \#grids < 1$, all outliers will be included within the decision boundary which will lead to failure in detecting Trojans. In practice, we found that a good value for ν can be obtained by setting $\nu \times$ #grids equal to about 10 to 15, which results in 10 or 15 outliers being kept outside the boundary.

- *Kernel function and parameters:* The hyperplane decision boundary does not always give the best result. To have a more flexible way of choosing decision boundary, we introduce the use of kernel functions. The idea is to map the

feature vectors from the input space F to a high-dimensional feature space H and use a hyperplane decision boundary in the high-dimensional space to separate most of the training samples from the origin. The decision boundary in the original input space, which is the pre-image of that hyperplane, can therefore take on many forms. The corresponding optimization problem can be written as follows:

$$\min_{\boldsymbol{\omega}\in H,\, \rho\in\mathbb{R},\, \boldsymbol{\xi}\in\mathbb{R}^m} \quad \frac{1}{2}\|\boldsymbol{\omega}\|^2 + \frac{1}{\nu m}\sum_{i=1}^{n}\xi_i - \rho \tag{3.8}$$

subject to

$$\boldsymbol{\omega}\cdot\Phi(\mathbf{x}_i) \geq \rho - \xi_i, \quad \xi_i \geq 0, \quad i = 1, 2, \ldots, m$$

where Φ is a map of feature vectors from input feature space F to high dimensional feature space H and other parameters are the same as Equation (3.6).

The label of an unknown vector $\mathbf{x}^*$ is given by:

$$f(\mathbf{x}^*) = \operatorname{sgn}(\boldsymbol{\omega}\cdot\Phi(\mathbf{x}^*) - \rho) \tag{3.9}$$

The problem is how to find such a mapping. Since the optimization problem above only requires dot products between feature vectors, we can design the mapping in such a way that the dot products in the high dimensional space can be computed in the original input space using a *kernel function* [96] $\mathbf{k}(\mathbf{x}, \mathbf{y})$ defined as follows. Note that $\mathbf{x}$ and $\mathbf{y}$ are two vectors in the original input space F.

$$\mathbf{k}(\mathbf{x}, \mathbf{y}) = (\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y})) \tag{3.10}$$

So instead of mapping our data via Φ and computing the dot products in a high dimensional space, we do it in one step, leaving the mapping completely implicit. In fact in the end we do not even need to know Φ, all we need to know is how to compute the dot products via a kernel function.

In [99], the author proposes four kernel functions such as gaussian and polynomial. Our results show that polynomial kernel function gives the best result. Thus we will use polynomial kernel function of the following form:

$$\mathbf{k}(\mathbf{x}, \mathbf{y}) = (\gamma \mathbf{x} \cdot \mathbf{y})^d \tag{3.11}$$

where γ is a parameter whose value can be decided by cross validation. In this paper, we use the default value in LIBSVM where $\gamma = 1/\#features$. d is the degree of polynomial. Too large d will cause the problem of overfitting while too small d will cause large training error. In practice, we found that $d = 5$ kept a good balance between training error and real error.

3.4.3 Final Classification

The final classification for each chip is decided by looking at the number of grids classified as Trojan-Inserted (TI) and their location. We do not classify a chip as TI if it has only a few sparse TI grids. Rather, we assume that in order for the chip to be classified at TI there must be at least n neighboring TI classified grids

in the chip. Neighboring grids can be defined within layers (horizontally adjacent grids) and between neighboring IC layers (vertically adjacent grids). This is based on the observation that malicious modifications tend to be continuous. They are either connected in some way on one layer or connected to some other malicious modifications through neighboring layers. The maximum number of connected negative grids gives us a notion of how large deviations from the golden layout there are in the chip under test.

3.4.4 Merits of the Proposed Approach

Our proposed approach is noteworthy for several reasons:

- In contrast to the naive approach (Section 3.3), we eliminate RE steps 4 and 5 which are not only very excessive, but also may require manual effort. In contrast, our approach is fully automated and considerably less expensive since we do not need to obtain an entire netlist for each chip. Furthermore, since our approach doesn't use the netlists for detection, we may also be able to detect parametric Trojans.

- Another interesting feature of our approach is how we divide the classification into smaller sub-problems. This allows parallelizable feature extraction and SVM classification of grids for very fast/distributed computations.

- Finally, our approach accounts for noise in the RE ICs in the SVM training phase. The noise margins we employ in creating the features allows us to intelligently distinguish between differences in the ICs caused by fabrication

and RE variations vs. systematic changes caused by an HT. Furthermore, ν-SVM makes our approach tunable (via ν parameter) to outliers that might exist in the training samples.

On one final note, we have only proposed using this our approach to build a golden model for other HT detection approaches because reverse-engineering is destructive. However, in the future, if less destructive methods are discovered to obtain images for one or more IC layers, our approach could also be applied to perform Trojan detection for malicious additions, deletions, and parametric changes. Our SVM-based approach would provide better coverage of HTs than other existing test-time methods (functional cannot detect parametric changes while side-channel analysis cannot detect small Trojans).

3.5 Trojan Detection Experiments and Results

3.5.1 Experiment Setup

Benchmarks. We tested our approach on 6 publicly available benchmarks(from ISCAS89 and ITC99). They were all synthesized using Cadence RTL Compiler with Synopsys 90nm Generic Library and placed and routed with Cadence Encounter. Reports from Cadence tool showed that these benchmarks have a gate count ranging from 57 gates for s27 to 122,559 for b18 (see Table 3.1).

Golden layout extraction. The golden layout was then obtained by GDSII file generated by Cadence Encounter tool. This file was later parsed by a Matlab script and used to generate a binary image for each layer. Each pixel in the image represents

Table 3.1: Benchmarks used in the experiments.

Benchmarks	#gates	Source	#training chips	Training Time(s)
s27	57	ISCAS89	20	121
s298	283	ISCAS89	5	175
s5378	3455	ISCAS89	1	351
s15850	10984	ISCAS89	1	1032
s38417	30347	ISCAS89	1	3055
b18	122559	ITC99	1	13054

5nm in reality. '1' in the image denotes there are structures at that position in the golden layout while '0' denotes that there is nothing in the golden layout.

Trojan insertion. For each benchmark, we implemented three kinds of malicious modifications. We randomly duplicated one component, deleted one component and selected one gate and doubled its width. We will refer to these three modifications as Trojan addition(TA), Trojan deletion(TD) and Trojan parametric(TP). We will refer to original Trojan-free design as TF. In our experiments, we simulated fabrication and reverse-engineering noise (see below) for 10 "sample" chips of each type (TF, TA, etc.). Note that in the modifications made to 10 sample chips of each type, the same gate was added, deleted or altered but the noise was different.

ICUT generation and noise. Using the same method as golden layout extraction, we generated binary images for these modified designs. To simulate process variations in fabrication and reverse-engineering, we added some random noise to these binary images. Figure 3.7 shows the noise. Those images with noise would be used as SEM images of ICUT.

Table 3.2: Chip classification accuracy rate averaged over ten trials.

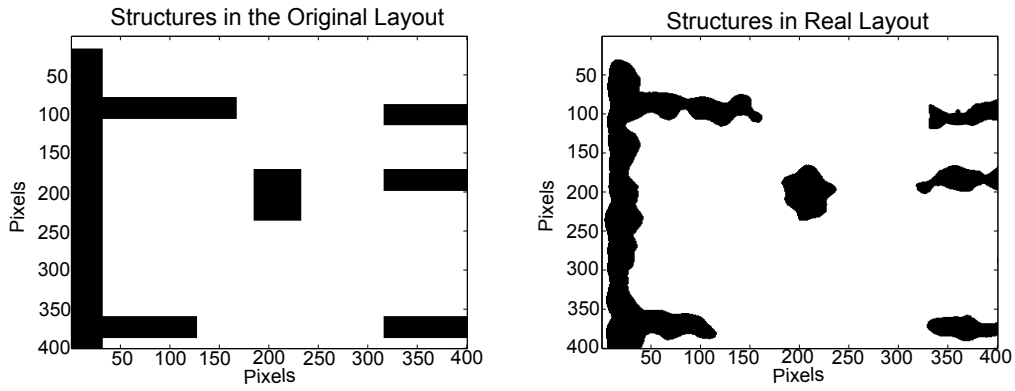
Benchmark	TF	TA	TD	TP
s27	100%	100%	100%	100%
s298	100%	100%	100%	100%
s280	100%	100%	100%	100%
s15850	100%	100%	100%	100%
s38417	100%	100%	100%	100%
b18	100%	100%	100%	100%

Table 3.3: Chip classification accuracy rate for ten s298 chips with varying ν.

ν	TF	TA	TD	TP
5e-4	100%	9%	6%	0%
2e-3	100%	98%	94%	82%
7e-3	100%	100%	100%	100%
2e-2	54%	100%	100%	100%
5e-2	0%	100%	100%	100%

SVM training. In our experiments, we use the LIBSVM [100] as the tool to solve ν-SVM optimization problem (Equation (3.8)).

Model and algorithm parameters. We used polynomial kernel function defined by Eqation (3.11) and chose degree $d = 5$. We used the default in LIBSVM $\gamma = 0.2$. We varied ν according to the number of grids per layer in the design so that $\nu \times \#$ grids is around 10. We set the grid size discussed in Section 3.4 to be 160×160 pixels (800×800 nm in reality), which is $20\lambda \times 20\lambda$. We chose the threshold on number of connected negative grids as $n = 2$. We set $d_{nm} = 8$ (note that by examining the



(a) Golden layout (b) Real layout

Figure 3.7: Simulation of process variations in reverse-engineering process

golden layout images and ones with noise, we could estimate that the value of noise margin d_{nm} defined in Section 3.4.1 is around 10 pixels). These parameters were chosen because they gave the best results according to our previous experiments.

Experiments and results recorded. We conducted four experiments. The first one was to test our method's ability to detect HTs using the above parameters. We selected 1˜20 chips (shown in Table 3.1) randomly from TF, TA, TD and TP chips and used them as training samples. The test samples consisted of 10 TF, 10 TA, 10 TD and 10 TP chips. Training phase was then conducted and test samples were classified as Trojan-free or Trojan-infested. To get a notion of the runtime of the algorithm, we record the training time for one case of each benchmark. Accuracy rate defined in Equation (3.12) was recorded for each case. This training and classification process was repeated for 10 trials and accuracy rate was averaged as shown in Equation (3.12). The difference between ten trials was samples chosen for training.

$$r_{acc} = \frac{\#\text{mislabeled chips}}{\#\text{chips under test}} \tag{3.12}$$

In the second, third and fourth experiments, we varied three different parameters, namely ν, grid size g, and noise margin d_{nm} for s298 benchmark while keeping everything else the same as the first experiment. In addition to accuracy rate, we also recorded the false negative/positive rate of classification of all the grids given by:

$$f_{-} = \frac{\#\text{false negative grids}}{\#\text{true positive grids} + \#\text{false negative grids}} \tag{3.13}$$

$$f_{+} = \frac{\#\text{false positive grids}}{\#\text{true negative grids} + \#\text{false positive grids}} \tag{3.14}$$

3.5.2 Results and Discussion

Results of the first experiment are listed in Table 3.2. They demonstrate that our method can detect three kinds of malicious modifications, including malicious addition, deletion and parametric changes, with high accuracy rate while recognizing Trojan-free chips reliably.

Results of the second experiment are shown in Table 3.3. Note that s298 has 1936 grids per layer for grid size $20\lambda \times 20\lambda$. The results show that when ν is very small ($\nu \times \#\text{grids} < 5$), we cannot detect Trojans very well because # of outliers identified in the training samples is small. Therefore most of outliers in training samples will be mislabeled as positive. In the classification phase, many negative samples whose feature vectors are similar to positive samples will be mislabeled, resulting in failure

Table 3.4: Average false negative rate f_-, false positive rate f_+ and chip classification accuracy r_{acc} for ten s298 chips of four kinds with varying grid size.

Gridsize	TF			TA			TD			TP		
	f_-	f_+	r_{acc}	f_-	f_+	r_{acc}	f_-	f_+	r_{acc}	f_-	f_+	r_{acc}
$10\lambda \times 10\lambda$	0.043%	-	78%	0.041%	5.8%	100%	0.039%	9.8%	100%	0.037%	1.7%	100%
$20\lambda \times 20\lambda$	0.037%	-	100%	0.041%	6.5%	100%	0.034%	14.5%	100%	0.035%	10.3%	100%
$30\lambda \times 30\lambda$	0.031%	-	100%	0.023%	5.2%	100%	0.027%	36.5%	100%	0.028%	19.7%	100%

Table 3.5: Average false negative rate f_-, false positive rate f_+ and chip classification accuracy r_{acc} for ten s298 chips of four kinds with varying noise margin values.

Noise Margin (Pixels)	TF			TA			TD			TP		
	f_-	f_+	r_{acc}	f_-	f_+	r_{acc}	f_-	f_+	r_{acc}	f_-	f_+	r_{acc}
0	0.42%	-	84%	0.31%	8.18%	100%	0.42%	82.8%	92%	0.33%	21.8%	100%
8	0.037%	-	100%	0.042%	7.52%	100%	0.037%	13.6%	100%	0.036%	10.6%	100%
16	0.036%	-	100%	0.071%	3.41%	100%	0.043%	16.9%	100%	0.071%	3.19%	100%
24	0.30%	-	96%	0.18%	6.65%	100%	0.28%	24.9%	96%	0.16%	11.6%	100%
30	0.81%	-	63%	0.28%	6.81%	100%	0.45%	83.7%	54%	0.36%	87.8%	18%

to detect Trojans. Detection of parametric Trojans is extremely hard because their modifications are very small and their feature vectors are very similar to Trojan-free grids. On the contrary, when ν is too large, the decision boundary surrounds those true positive training samples tightly and even some positive samples may be mislabeled. This leads to detection of all the Trojans but also causes false alarms (which is indicated by a failure to detect all Trojan-free chips reliably). The results suggest that $\nu \times \#grids$ should be around 10 to 15 to get the best result.

Results of the third experiment are shown in Table 3.4. The results reveal that as grid size increased from $10\lambda \times 10\lambda$ to $30\lambda \times 30\lambda$, the false negative rate decreased. This is because when the grid size is small, the denominator in Equation (3.1), (3.2) and (3.3) tend to be small, causing it to be more sensitive to noise. Furthermore, when the grid size gets smaller, total number of grids will be larger but ν remains the same, so there may be more grids mislabeled as outliers. When grid size is equal to $10\lambda \times 10\lambda$, more false negative grids existed and were connected, giving rise to false alarm in 4 out of 10 TF chips. On the other hand, when grid size increased, the false positive rate increased. This is because when grid size is large, the method is less sensitive to noise and small modifications within a grid. In the worst case, there may be so many false positive grids that negative grids are not connected, leading failure to detect Trojans. Moreover, when the grid size gets larger, the number of grids that contain malicious modifications gets smaller. When this number is equal to one, we cannot detect the Trojan at all. Thus, large grid size does not work for small Trojans. Therefore, we concluded from the results that grid size could neither be too small or too large.

Results of the fourth experiment are listed in Table 3.5. Results showed that our approach worked the best when the value of noise margin was close to its true value(when it was 8). However, when the value of noise margin deviated from its true value slightly (when $d_{nm} = 16,\ 24$), the results were only slightly worse. This is actually good because we do not need to accurately estimate the true value of noise margin, which is almost impossible to be done precisely. We just need to pick a reasonable guess and our approach would work.

In summary, the results above show that our proposed method can detect HTs with high accuracy rate if parameters are chosen correctly. The SVM modeling parameter ν should be decided according to the number of grids per layer. When the grid size $g = 20\lambda \times 20\lambda$ and noise margin d_{nm} is close to its true value, results are the best. However, if d_{nm} deviates a little from its true value, results are only slightly worse.

3.6 Design-for-Security: Motivation

Motivation. As shown in Section 3.5, our detection approach can achieve very high detection accuracy even with one training chip and does not require the existence of a golden chip. Thus, we will use it as our post-manufacturing hardware Trojan detection method. We are interested in finding any design-time strategies that can make detection of Trojans easier using that method.

As a motivating example, we choose one benchmark, s298, from ISCAS89 [101]. We select two subsets of standard cells from Synopsys 90nm generic library, namely {DFFX1, INVX4, NAND2X1, NOR2X0} and {DFFX1, INVX2, NAND2X4, NOR2X2}. We synthesize the benchmark using these two subsets respectively, and generate two synthesized designs. For each design, we randomly select one standard cell from it as attack target and make several malicious modifications including structure deletion and addition to that target at the layout level. These modified designs will serve as the Trojan-infected designs. In total, for each design, random selection of the attack target is done 10 times, and each time 10 different modifications are applied to it,

Table 3.6: Trojan miss rate, false positive rate, area and leakage power values of two benchmarks. Numbers in parenthesis are ratios of values in two designs.

	TMR (%)	**FPR (%)**	**Area (μm^2)**	**Power (nW)**
Design 1	33.14 (1.41X)	0.35	1167	6622
Design 2	23.55	0.43 (1.21X)	1403 (1.20X)	9122 (1.38X)

making the total number of Trojan designs 100 for each synthesized design. We then apply training and classification techniques introduced in [42] to both designs. The training is done on 5 Trojan-free chips. After classification of each grid in the IC is done, Trojan Miss Rate (TMR) and False Positive Rate (FPR) defined below averaged over 100 Trojan designs are recorded. The area and leakage power are also reported in Table 3.6. Note that no timing constraints are specified during the synthesis.

$$TMR = \frac{\#\text{grids with malicious modifications and are } \textbf{NOT} \text{ detected}}{\#\text{grids that have malicious modifications}} \tag{3.15}$$

$$FPR = \frac{\#\text{grids without malicious modifications but are mislabeled}}{\#\text{grids that have no malicious modifications}} \tag{3.16}$$

This table shows that two designs have significant difference in Trojan miss rate. In general, we can detect 41% more malicious modifications made to the second design than those made to the first design. However, this does not come for free. Design 2 also has a higher false positive rate and incurs 20% overhead in area and leakage power. We summarize our findings from the above motivating example:

- Malicious modifications made to some standard cells in a given library may be easier to be detected than those made to other standard cells. Put in another way, some standard cells are more sensitive to malicious modifications and are

thus favored over others in terms of Trojan detection.

- Though design 2 has a higher false positive rate (FPR) which is 0.43%, it is negligible. Moreover, since we intend to use the technique to identify golden chips, what is more important is to decrease the Trojan miss rate, which is also the false negative rate. Because false positive rate and false negative rate cannot be decreased at the same time, we will neglect FPR and only focus on TMR.

- The increase in Trojan detection accuracy usually leads to extra area/power overhead.

This motivates us to find a 'best' subset of standard cells such that if the circuit is synthesized on it, detection of Trojans using [42] will be the most accurate while area/power/timing overheads are acceptable. However, we do not want to write our own EDA tool because it is tedious and error-prone. Instead, we are interested in modifying existing EDA tool flow to make it security-aware so that our approach can be readily used. We will define this problem formally next.

Problem Definition. Given a technology library l consisting of a set of n standard cells such that $l = \{C_1, C_2, \ldots, C_n\}$, a circuit in the form of RTL code or netlist denoted by d, an area upper bound A_{up}, leakage power upper bound P_{up} and some timing constraints $t_1, t_2, \ldots$, find a subset of l, namely $l_s = \{C_{s_1}, C_{s_2}, \ldots, C_{s_i}\}$, such that when we synthesize d on l_s, malicious modifications made to it will be the easiest to be detected (among all possible subsets l_s) given that the following constraints

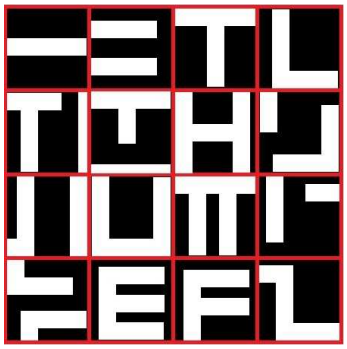


Figure 3.8: 16 Primitive Structures

Standard Cell	CTMR	TMR
NAND2X0	36.08%	27.32%
NAND2X1	43.08%	39.33%
NAND2X2	38.31%	32.83%
NAND2X4	33.18%	25.38%

Table 3.7: $CTMR$ and actual Trojan miss rate of some standard cells

are met:

$$l_s \text{ is universal, } t_1, t_2, \ldots \text{ are satisfied}$$
$$\text{area} \leq A_{up}, \text{ leakage power} \leq P_{up} \tag{3.17}$$

Challenges. Note that the above problem is actually an optimization problem where the possible solution space is the power set (the set of all subsets) of l, the objective is to maximize Trojan detection accuracy and the constraints are defined by (3.17). We argue that the following challenges have to be addressed before solving the above problem.

- The objective function is vague. We have to define mathematically Trojan detection accuracy.

- In order to find the best subset of standard cells, a characterization of each standard cell has to be done.

- The possible solution space is exponential. For example, saed90nm library has 340 standard cells. Thus, its power set contains $2^{340} = 2.23 \times 10^{102}$ elements. Moreover, for each possible solution, we need to run the entire synthesis flow to estimate the area, power, etc. thereby making the approach highly complicated. Obviously, exhaustive search is computationally prohibited and heuristics are needed.

3.7 Design-for-Security: Implementation

3.7.1 Characterization of Each Standard Cell

Before proposing our method of standard cell selection, we first investigate what causes the difference in sensitivity to malicious modifications between different standard cells. Reverse-engineering based Trojan detection approaches such as [42] usually extracts several features like centroid difference, area of difference, etc. from each grid. Since different primitive structures (such as 'T' shape, 'L' shape, 'F' shape, etc.) in a grid have different centroid patterns, they may have different sensitivities to malicious modifications. Therefore, in order to characterize the sensitivity of each standard cell, we have to characterize its building blocks, which are these primitive structures first.

We identify 16 possible structures (shown in Figure 3.8) in a grid as primitives.

Note that though these 16 primitives are not exhaustive, our experiments show that over 90% of the grids contain exactly one of them. Therefore, they are a good representative of primitive structures. We will synthesize several benchmarks, apply gridding and use the method in [42] to train one classifier for each benchmarks. We say a grid is of type $i (1 \leq i \leq 16)$ if it contains only i^{th} primitive. Otherwise we say it is of type 17. After training, we make some malicious deletions and additions to these benchmarks and use the classifier to classify each grid. We then calculate the Trojan Miss Rate (TMR) defined by Eqn (3.15) for each type of the grid respectively. We let p_i $(1 \leq i \leq 17)$ denote the TMR of grid type i.

After the characterization of each primitive structure is done, given a technology library, we can apply the same gridding to each standard cell and label each grid as 1-17 using the same way as above. Say that there are n_i grids of type i $(1 \leq i \leq 17)$ in the standard cell. We define below CTMR (Cell Trojan Miss Rate) which measures the average probability that a malicious modification is not detected if it happens at one random grid of the cell.

$$CTMR = \frac{\sum_{i=1}^{17} n_i \times p_i}{\sum_{i=1}^{17} n_i} \tag{3.18}$$

By definition, the smaller $CTMR$ a cell has, the easier it is to detect malicious modifications made to it. To see this, we measure the actual Trojan miss rate of all NAND2 standard cells by synthesizing a NAND2 gate using these standard cells respectively, making some malicious modifications to the synthesized design and calculating TMR defined by Eqn (3.15). We also calculate $CTMR$ of these standard cells and list them in Table 3.7. The results indicate that $CTMR$ is a great metric

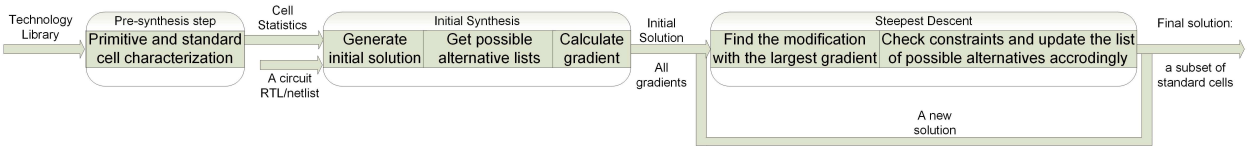


Figure 3.9: Overall Block Diagram of Our Proposed Method

for cell's sensitivity to malicious modifications.

3.7.2 Mathematical Description of The Objective Function

Now that we have a good metric for cell's sensitivity to malicious modifications, we can define a design's sensitivity accordingly. Suppose a circuit is synthesized on a technology library and the synthesized design contains k_i $(1 \leq i \leq n)$ standard cell C_i. Let $l = \{C_1, C_2, \ldots, C_n\}$. Then we calculate this synthesized design's Trojan miss rate (DTMR) as below:

$$DTMR(l) = \frac{\sum_{i=1}^{n} k_i \times CTMR_i}{\sum_{i=1}^{n} k_i} \tag{3.19}$$

where $CTMR_i$ is the Trojan miss rate for cell C_i (defined by Eqn (3.18)). We use $DTMR$ as the metric to measure the design's sensitivity to malicious modifications. The smaller $DTMR$ a design has, the easier it is to detect malicious modifications made to this design. Note that when the circuit is fixed, $DTMR$ is only determined by a subset of standard cells, denoted by l, on which the circuit is synthesized. Thus, we write $DTMR$ as a function of l.

We can reformulate the problem in Section 3.6 as:

$$min_{l_i \in 2^l} \;\; DTMR(l_i) \tag{3.20}$$

subject to constraints defined in Eqns (3.17). Here 2^l means the power set (the set

of all subsets) of the given technology library l. The subset of standard cells denoted by l_{opt} that leads to the optimal value of the above optimization problem is what we want. We will call l_{opt} the *solution* to the above problem.

3.7.3 Steepest Descent Method

As stated in Section 3.6, an exhaustive search is computationally prohibited. Heuristics are needed to reduce the search space and find a near-optimal solution. We will instead, adapt the idea of the steepest descent method (SDM) to solve the above problem. To find a local minimum of function f, the general SDM works as follows:

1. Find an initial solution x_0 and set $x_{old} = x_0$.

2. Calculate the gradient at the current solution and construct a new solution $x_{new} = x_{old} - \lambda \nabla f(x_{old})$ where λ is the step size that controls the convergence rate and should be kept relatively small.

3. If $f(x_{new}) < f(x_{old})$ then we set $x_{new} = x_{old}$ and go back to step 2. Otherwise, a local minimum point x_{old} has been found.

Note that the performance of SDM depends heavily on the initial solution. However, when the initial solution and the step size is chosen carefully, it can lead to very good solutions. Therefore, we will adapt its idea to solve our problem. But applying SDM to our problem has the following challenges, some of which are due to the nature of the method and some are unique to our problem.

- The performance of the SDM relies heavily on the choice of the initial solution. How to select the initial solution should be carefully investigated.
- The SDM is based on recursively updating the current solution to get a better solution. How to do this efficiently and effectively has yet to be examined.
- The SDM only works where the function to be optimized is differentiable. However, in our problem, it is a discrete function. We have to define the gradient accordingly.

We will address these challenges in the following text.

3.7.4 Our Proposed Algorithm

The overall proposed flow is shown in Figure 3.9. We will explain each step as well as the intuition behind the step.

Choice of initial solution. We can make up an arbitrary initial solution, which is to choose an arbitrary subset of standard cells. However, there is no guarantee that synthesizing the circuit d on this subset will meet area/timing/power constraints. In fact, the chance is very low. However, we can obtain an initial solution in a better way. We synthesize the design on the whole library applying all timing constraints. This will give us a list of standard cells used together with area and leakage power of the design. This is a baseline design and its area and leakage power are the minimum values we can get given that all timing constraints are met. So it must be one feasible solution (if this design does not meet the area and power constraints, then no design will meet those constraints and no feasible solution exists) and is a

very good point to start with. We will set the list of standard cells used in that synthesis as our initial solution.

How to modify the current solution. In the context of our problem, the modification is done by adding/removing several standard cells into/from the current solution. This is where the exponential complexity comes in because there are exponential numbers of ways of adding/removing standard cells to/from the old solution. To reduce the computation complexity, we make a key observation here.

Typically in a library, there are multiple standard cells with different drive strength that implement exactly the same logic function. For example, AND2X1, AND2X2, AND2X4 all implement logic AND function of two operands: $Out = In1\&In2$ but with increasing drive strength. We group all standard cells that implement the same logic function but with different drive strength into one group. For each standard cell C_i, we say standard cell C_i' is a *possible alternative* of C_i if it satisfies the following three properties: 1) It is in the same group with C_i. 2) It has a larger drive strength than C_i. 3) It has a lower $CTMR$ defined by Eqn (3.18) than C_i. We call all possible alternatives of C_i its *list of possible alternatives* and we limit the way we modify the old solution such that a standard cell in the old solution can only be replaced by another standard cell in its *list of possible alternatives*. If such list is empty, that standard cell is fixed in the solution and can never be moved out from the old solution. Such limitations have the following advantages:

- By ensuring that a standard cell is replaced with another one that implements exactly the same function, we can perform incremental synthesis in EDA which

can obtain power/area/timing information much faster than synthesizing a design from scratch. The correct functionality of the new design is also trivially guaranteed.

- Since all standard cells in the possible alternative list have a greater drive strength, enough drive strength originally required by the design is guaranteed.

- By limiting the replacement of one standard cell with only another standard cell in its *list of possible alternatives*, the search space is cut significantly. By the third property of the possible alternative, the search space is further reduced without affecting the optimality of the solution.

- It can be easily implemented, fully automated and 100% compatible with current flow of a commercial CAD tool.

Note that when we impose the above limitations, the modifications to the current solution can be done additively. This means that instead of adding/removing multiple standard cells from the design in one shot, we can always limit each modification to replacing only one standard cell say C_i with another standard cell C_i^j in its *list of possible alternatives*.

Gradient calculation. Since each modification to the current solution is done by replacing one standard cell with another one that is in its list of possible alternatives, we only need to define the gradient for each standard cell in all lists of possible alternatives of the initial solution. The gradient should assess the quality of the modification. Due to the discrete nature of our problem, gradient has to be defined

in a different way. The detailed steps to obtain the gradient is defined below:

1. Let $l_0 = \{C_1, C_2, \ldots, C_k\}$ denote the initial solution. Let p_0, a_0, p_0 denote the leakage power, area, and timing slack obtained after initial synthesis.

2. For each standard cell in l_0, we identify its list of possible alternatives. We replace a standard cell C_i with one of its possible alternative C_i^j and keep all the other standard cells in l_0 unchanged. We call this new subset of standard cells l_i^j. We resynthesize the design on l_i^j and obtain new leakage power p_i^j, area a_i^j and timing slack t_i^j. Let $DTMR(l_0)$ and $DTMR(l_i^j)$ denote the design's Trojan miss rate (defined by Eqn (3.19)) when synthesizing the circuit on l_0 and l_i^j respectively. We define the following quantity first:

$$\Delta DTMR(l_i^j) = DTMR(l_0) - DTMR(l_i^j)$$
$$\Delta t(l_i^j) = \frac{t_0 - t_i^j}{t_0},\ \Delta a(l_i^j) = \frac{a_i^j - a_0}{a_0},\ \Delta p(l_i^j) = \frac{p_i^j - p_0}{p_0}$$

   We then define the gradient as follows:

$$\nabla DTMR(l_i^j) = \frac{\Delta DTMR(l_i^j)}{\Delta t(l_i^j) + \Delta a(l_i^j) + \Delta p(l_i^j)} \tag{3.21}$$

Intuitively, this quantity measures how much improvement in $DTMR$ we can get with 1% overhead in power, area and timing slack by replacing C_i with C_i^j in l_0 to form the new library l_i^j. The greater this gradient is, the more improvement in $DTMR$ such replacement can yield given that the overhead stays the same.

Overall Algorithm. Putting all things together in the framework of the steepest descent method, we detail the steps of our algorithm below:

1. Identify and characterize primitive structures. Then characterize each standard cell in a given technology library. Note that for a given technology library, this step only needs to be done once.

2. Synthesize the design using the whole technology library and obtain the subset of standard cells used denoted by l_0. This is our initial solution. We set $l_{old} = l_0$.

3. Identify the list of possible alternatives for each of the standard cells in l_0. Obtain the gradient defined by Eqn (3.21) for every standard cell in every list of possible alternatives.

4. Find in all lists of possible alternatives the standard cell with the largest corresponding gradient, say C_i^j which is j^{th} standard cell in the possible alternative list of C_i.

5. Replace C_i with C_i^j in l_{old} and let l_{new} denote the new subset of standard cells. Synthesize the design on l_{new} and check if area/power/timing constraints are met. If so, set $l_{old} = l_{new}$ and set the possible alternative list of C_i to empty. If not, remove C_i^j from the possible alternative list of C_i and keep l_{old} unchanged.

6. Repeat step 4-5 until all lists of possible alternative are empty. l_{old} is what we want.

3.7.5 Merits of Our Approach

Our proposed method is noteworthy for the several reasons:

- It works with the current design flow of a CAD tool. This means we do not need to write our own CAD tool. All we need is to implement our algorithm as a script and the CAD tool will generate a security-aware synthesized design.

- We cut the search space significantly while still being able to find a rather near-optimal solution (as indicated in experimental results in Table 3.9). Also, our definition of gradient allows us to keep a good balance between the optimality of the solution and overheads in timing/area/leakage power.

- We avoid synthesizing designs from scratch by using incremental synthesis. This saves the algorithm run-time greatly. Furthermore, if we assume the incremental synthesis time is a constant, then after initial synthesis, our algorithm run-time is only linearly dependent on the number of standard cells in the technology library and is independent of the circuit size. Thus, our algorithm is efficient and scalable.

3.8 Design-for-Security: Experiments and Results

3.8.1 Experiment Setup

Benchmarks. We test our approach on 8 publicly available benchmarks (from ISCAS89 [101] and trustHub [102]). They are all synthesized using Cadence RTL Compiler with Synopsys 90nm generic library. Reports from Cadence tool show that these benchmarks have gate counts ranging from 69 for s298 to 175456 for AES (shown in Table 3.8). The detailed description of each benchmark is listed below.

Table 3.8: Benchmarks used in the experiments.

Benchmark	Source	#Gates	Clock Frequency
s298	ISCAS89	69	25MHz
s5378	ISCAS89	728	25MHz
s15850	ISCAS89	1968	25MHz
s38417	ISCAS89	5066	25MHz
AES	TrustHub	175456	500MHz
b19	TrustHub	45150	25MHz
MC8051	TrustHub	6927	50MHz
XGE_MAC	TrustHub	60222	156.25MHz

Applying constraints. We set the load capacitance to 100pF on all output ports for all benchmarks. We set the input/output delays to 200ps. The clock frequency of each benchmark is listed in Table 3.8. For each benchmark, we apply the above timing constraints (no timing violations are tolerated in our experiments) and synthesize the design using the Synopsys 90nm generic library. We can then obtain the area and leakage power after synthesis is done. We then set the allowed overhead for leakage power and area to 30%, which means the leakage power and area should not exceed their original values by more than 30%.

Baseline design and security-aware design generation. For each benchmark, we synthesize the circuit using the whole library with all timing constraints applied. We call this design the baseline design. It represents a design that is optimized in area and meets all timing constraints but lacks any security concerns. We then apply our algorithm and obtain a list of standard cells. We resynthesize the circuit

on this list of standard cells and call it security-aware design. We will use Cadence Encounter tool to place and route both designs.

Trojan insertion. For each benchmark, for both baseline design and security-aware design, we randomly choose a standard cell as our attack target and make 4 malicious modifications to it (2 addition of structures and 2 removal of structures). The random selection of attack target is done 15 times and in total 60 Trojan-inserted chips are generated.

Experiment conducted and results recorded. For each benchmark, for both baseline design and security-aware design, we train SVM over 5 Trojan-free chips for s298 and 1 Trojan-free chip for all other benchmarks and use the classifier to classify the 60 Trojan-inserted chips. The detailed training and classification steps as well as parameter selection follow the work in [42] with one small distinction. Instead of classifying the whole IC, we only generate the labels for each grid in the IC. We then calculate the Trojan miss rate defined in Eqn (3.15) for both designs averaged over 60 chips. The difference of Trojan miss rate between baseline design and security-aware design is then calculated and the ratio of the difference to the Trojan miss rate of the baseline design is reported as TMR improvement. We also report the leakage power and area overhead of security-aware design compared with baseline design. The results are listed in Table 3.9. The algorithm run-time as well as the time consumed for the initial synthesis is also reported. Note that the algorithm run-time does not include the pre-synthesis step shown in Figure 3.9.

Table 3.9: TMR improvement, leakage power overhead (LO), area overhead (AO), algorithm run time (AT) and the original synthesis time (OT) of all benchmarks

Benchmark	TMR Improvement	AO	LO	OT(s)	AT(s)
s298	20.93%	10.09%	17.67%	16.4	283.9
s5378	13.04%	7.91%	16.05%	31.3	663.6
s15850	15.23%	6.21%	13.94%	44.4	906.1
s38417	16.00%	4.00%	8.58%	103.4	990.4
AES	23.45%	16.68%	28.21%	2302	9941
b19	23.21%	8.51%	29.74%	759.1	3841
MC8051	15.62%	7.70%	21.99%	204.3	10010
XGE_MAC	7.48%	1.85%	5.60%	2031	9055

3.8.2 Results

From Table 3.9, we can see that using our method , we can detect on average 16.87% more malicious modifications with only 7.87% area overhead and 17.72% leakage power overhead. Note that Cadence RTL Compiler always optimizes the area given that all timing constraints are met, making the area overhead very small compared with leakage overhead. We also note that the our algorithm's performance depends heavily on the timing requirements of the design. If a design's timing requirements are easily met, then our algorithm can improve TMR greatly. Examples are AES and b19. However, if a design's timing requirements are very hard to meet, then the improvement in TMR is limited. The example is XGE_MAC benchmark which implements MAC functions for 10Gbps operation. The reason is that when

timing requirements are hard to meet, each standard cell will have only very few possible replacement alternatives because replacement is very likely to cause timing violations. Therefore, the search space for our algorithm is small and the improvement in TMR is limited. Nevertheless, we still can detect 7.48% more Trojans with only 1.85% overhead in area and 5.60% in leakage power in the latter case. This proves the efficacy of our algorithm.

3.9 Conclusion

In this chapter, we propose a novel reverse-engineering based approach without generating a gate or transistor netlist. Our approach adapts one-class ν-SVM to the HT detection problem. The experimental results on several publicly available benchmarks show that our method can achieve very high Trojan detection accuracy. We also investigated the impact of some modeling and algorithm parameters on the accuracy rate. Our method is efficient in storage space, does not require the existence of golden chip and is robust to variations in fabrication and reverse-engineering process. We also propose a novel design-time strategy to aid test-time Trojan detection. Experimental results on real benchmarks showed that using our design-time strategy, we can detect on average 16.87% more Trojans with only 7.87% area overhead and 17.72% leakage power overhead. Our method is fully automated, can easily fit into the current design flow of IC and thus is very promising.

Chapter 4: Hardware Countermeasures Against Timing Side-Channel Attacks on Caches

4.1 Introduction

Recent studies on cache-based timing side-channel attacks show an urgent need for thorough investigations of security issues in CPUs. [2,3,61,103,104] show how to perform timing side-channel attacks on the data cache to leak the full key of many wide-used encryption algorithms, such as AES, RSA, etc. Attacks on the instruction cache [58] and branch prediction buffer [5] have also been demonstrated successful. Other than targeting at the cryptographic cipher, [4] demonstrates that keystroke information of a user program can be leaked. [56] shows how timing side-channel leakage can be exploited to attack address space layout randomization (ASLR), which is a commonly used technique to circumvent buffer overflow attacks. These new results on timing side-channel attacks demonstrate that attacks can happen at any level/type of the cache. They also demonstrate that these attacks can not only leak system's critical information, but also break some defense mechanisms (such as ASLR) and serve as the foundation for further attacks (e.g., buffer overflow attacks). Therefore, they are extremely dangerous. Though hardware countermeasures based

on cache partitioning [72], randomization [73,76], and securing the timing source [79] have been proposed, they all have moderate to heavy performance overhead.

With the slowdown of technology scaling in recent years, 3D integration stands out as the most promising alternative to continue the area shrinking and performance gain. In a 3D CPU, caches and even main memory are stacked vertically on top of the processors. The vertical interconnection is achieved using through-silicon vias (TSVs), which are much shorter than a 2D connection. The reduced wire length naturally results in delay reduction and consequently performance improvement [105]. Also due to large number of TSVs and extra layers of space, we can have more bandwidth and more cache and memory on chip. The availability of more intricate functional resources (due to 3D) as well as the associated reduction in timing and increase in bandwidth, can be used for mitigating the impact of timing side channel attacks. In this chapter, we investigate several intricate architectural techniques which exploit the availability of 3D integration for mitigating timing side-channel attacks. The major contributions of the work in this chapter are:

- We investigate the following parameters of 3D CPUs that impact both performance and security: more available resources, data migration schemes and cache-memory address mapping. We explore how to design these parameters to mitigate side-channel attacks while keeping their impact on performance minimum.

- We put together a simulation framework which quantifies the performance and security implications of the proposed countermeasures. We test our pro-

posed countermeasures against four widely-applied timing side-channel attacks: access-driven attacks on L1, access-driven attacks on last level of cache, time-driven attacks, and branch prediction attacks.

- The experimental results show that our proposed techniques can reduce the timing side-channel information leakage in all four attacks significantly while still achieving more than 20.43% performance gain over a 2D baseline processor.

It is important to note that the driving force of 3D integration is not security, but high performance and low power. However, our work shows that 3D integration also offers inherent security benefits and enables many new defense mechanisms that would not be practical in 2D. Our work is compatible with the ongoing trend of transition from 2D to 3D and enables designers to take security into account when designing future cache using 3D integration technology.

The rest of the chapter is organized as follows. Section 4.2 gives a brief introduction to cache-based timing side-channel attacks, related countermeasures and 3D CPUs. Section 4.3 introduces the attack model we are trying to defend against and investigates the benefits and new opportunities that 3D integration can bring us. Section 4.4 explains our proposed techniques in detail and Section 4.5 evaluates their performance and security. Section 4.6 discusses the merits of our proposed designs. Finally, Section 4.7 concludes the paper.

Table 4.1: Lookup Table Accesses in the First Round of AES. $x_i = p_i \oplus k_i$

Lookup Table	Access Index	Lookup Table	Access Index
T_0	x_0, x_4, x_8, x_{12}	T_1	x_1, x_5, x_9, x_{13}
T_2	x_2, x_6, x_{10}, x_{14}	T_3	x_3, x_7, x_{11}, x_{15}

4.2 Preliminary

4.2.1 Lookup Table-based Modern Cipher Implementation

For efficiency purposes, modern cryptographic ciphers are implemented as key-dependent lookup tables. We use the first round AES encryption (128-bit version) as an example to illustrate how it works [3].

During system initialization, four tables T_0, T_1, T_2, T_3, each containing 256 4-byte words, are precomputed following the steps in [106]. Given a 16-byte plaintext $\mathbf{p} = (p_0, \ldots, p_{15})$, the first round of encryption under the key $\mathbf{k} = (k_0, \ldots, k_{15})$ computes a 16-byte intermediate state by accessing the lookup tables shown in Table 4.1. The key observation relevant to this paper is that the access index to these tables are dependent on the key. Specifically, the index to lookup tables in the first round is the exclusive-or of the key and the plaintext. Therefore, leaking the access patterns to these lookup tables directly leaks the key.

4.2.2 Cache Timing Side-Channel Attacks on Modern Ciphers

Mathematically, we define predicate $Q(\mathbf{p}, y) = 1$ iff. the encryption of plaintext $\mathbf{p}$ (under the unknown key $\mathbf{k}$) accesses cache set y. $Q(\mathbf{p}, y)$ cannot be measured

directly. Instead, the attacker learns it based on some noisy timing measurements defined by $M(\mathbf{p}, y)$. Then the attacker can deduce the key using measurements M (details will be given shortly). The exact definition of M will vary, but it should approximate $Q(\mathbf{p}, y)$ in the following sense:

Lemma 4.2.1 *the expectation of* $M(\mathbf{p}, y)$ *should be larger when* $Q(\mathbf{p}, y) = 1$ *than when* $Q(\mathbf{p}, y) = 0$.

Prime+Probe attack [3]. The attacker runs his program on the same core with the victim process and they share some levels of cache (e.g., L1 cache). The attacker repeatedly performs the following steps: 1) Prime: the attacker fills the entire cache with his own data. 2) Idle: the attacker runs victim process with a chosen plaintext. 3) Probe: the attacker measures the time it takes to load each set of his data. The reload time of cache set y under the chosen plaintext $\mathbf{p}$ is his measurement $M(\mathbf{p}, y)$. The measurement is then normalized by subtracting the average timing of its cache set. If $Q(\mathbf{p}, y) = 1$, then the attacker's data will be evicted from the cache (see Figure 2.1), which causes cache misses for the attacker and results in a longer load time in the Probe phase (the load time for set2 and set4 will be longer in Figure 2.1), thus a larger $M(\mathbf{p}, y)$. This satisfies lemma 4.2.1. After obtaining measurements using many chosen plaintexts, the attacker can learn the partial information about i^{th} byte of the key k_i, in the one-round attack, by testing each candidate values $\tilde{k}_i$ in the following way (We only show the case for k_0. However, it is applicable to other bytes of the key).

Let $y(\mathbf{p}, b)$ be the cache set that $T_0[p_0 \oplus b]$ maps to, where p_0 is the first byte of the plaintext $\mathbf{p}$. From Table 4.1, we know that $y(\mathbf{p}, k_0)$ will always be accessed during the first round of the encryption. Thus $Q(\mathbf{p}, y(\mathbf{p}, k_0)) = 1$ for all chosen plaintexts $\mathbf{p}$. If $\tilde{k}_0 \neq k_0$, $Q(\mathbf{p}, y(\mathbf{p}, \tilde{k}_0))$ may or may not be 1. With Lemma 4.2.1, this means that the expectation of $M(\mathbf{p}, y(\mathbf{p}, \tilde{k}_0))$ should be the largest when $\tilde{k}_0 = k_0$. We define the **measurement score** for the candidate value $\tilde{k}_0$ after encrypting N random chosen plaintexts, denoted by $score(\tilde{k}_0)$, as the average of $M(\mathbf{p}, y(\mathbf{p}, \tilde{k}_0))$:

$$score(\tilde{k}_0) = \sum_{i=1}^{N} \frac{M(\mathbf{p}^i, y(\mathbf{p}^i, \tilde{k}_0))}{N} \tag{4.1}$$

where $\mathbf{p}^i$ denote the i^{th} plaintext. By calculating the measurement score for each candidate value and finding the argmax, the attacker knows k_0. However, in reality, since multiple lookup table entries are mapped to the same cache set, some candidate values will have the same measurement score. This means that only partial information of k_0, denoted by $\langle k_0 \rangle$ can be leaked. We perform the attack on Gem5 simulator (the detailed configurations used can be found in Section 4.3) with 300 chosen plaintexts. In our settings, 16 table entries will be mapped to the same cache set, therefore $\langle k_i \rangle$ will be the highest 4 bits of key k_i. The measurement score for deducing $\langle k_0 \rangle$ is shown in Figure 4.1. Based on Figure 4.1, the attacker concludes that $\langle k_0 \rangle = 8$ since the highest peak appears at 8.

Berntesin's Attack: The attacker will rely on the reuse of the cached data within the victim process. Bernstein [2] initiated the work in this area. The basic idea is to exploit the impact of the reused cached data on the total execution time of the encryption. As shown in Table 4.1, during the first round of the AES encryption, one

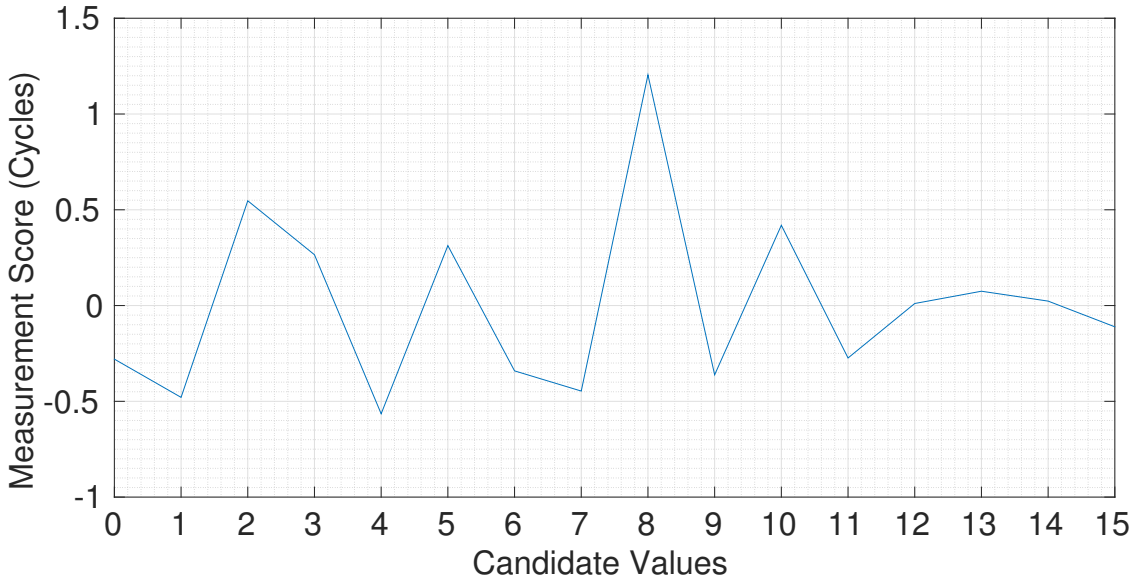


Figure 4.1: Measurement score for $\langle k_0 \rangle$ in 2D cache configurations.

table is accessed using the index $x_i = p_i \oplus k_i$, where p_i and k_i are the i^{th} byte of the plaintext and key respectively. The corresponding data is then cached and may be reused later. Thus, the entire encryption time t can be affected by each of the values x_i based on some internal cache contentions. To carry out this attack, the attacker first runs the encryption on his own machine (that has the same configurations with the target machine) with a chosen key $\mathbf{k}^{chosen}$ and a large volume of plaintexts, he records the time for each encryption. Then he calculates the average encryption time $t(i, x)$ for each byte $i, i = 0, 1, \ldots, 15$ and each possible value of $x = p_i \oplus k_i^{chosen}$. Then he performs similar steps on the target machine with the unknown key $\mathbf{k}$ to collect the average encryption time, $u(i, j)$ for each byte i and each possible value of $j = p_i$. Ideally:

$$t(i, x) = u(i, j), \text{if } x = k_i \oplus j \tag{4.2}$$

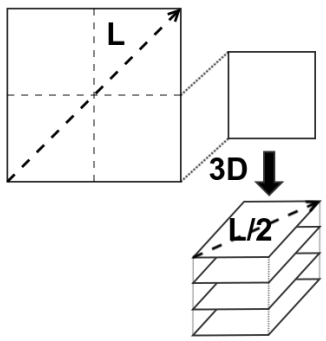


Figure 4.2: Wire-length reduction achieved in 3D cache configurations: using four stacked layers of cache, the interconnect length reduces by 50%.

Then, for each candidate value $\tilde{k}_i$ of key k_i, cross-correlation is calculated as:

$$corr(\tilde{k}_i) = \sum_{m=0}^{255} t(i,m) \cdot u(i, m \oplus \tilde{k}_i) \tag{4.3}$$

Due to Eqn (4.2), $corr(\tilde{k}_i])$ is the largest when $\tilde{k}_i = k_i$. Therefore, after taking enough samples, the attacker can calculate the cross-correlation and the $argmax(corr)$ leaks the key.

Though it requires large amount of samples to be collected, Bernstein's attack [2] is a generic attack and is widely applicable since it is extremely difficult to achieve fast constant-time software. Timing variation exists due to the need of memory hierarchy, which is inevitable in the current computer system. Therefore, this kind of attack is extremely hard to defend against. Similar attacks have been proposed in [69, 70]

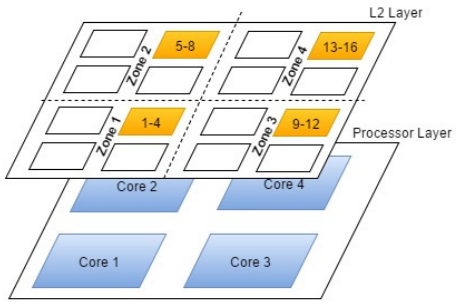


Figure 4.3: One NUCA L2 Configuration.

4.2.3 3D CPUs

Offering shorter wire-length and larger bandwidth, 3D integration has been viewed as a promising alternative of technology scaling to continue the performance improvement. Though 3D integration has its own shortcomings such as low yield and reliability issues. Researchers have proposed various techniques to overcome them [107–110]. Over the last decade, 3D CPUs have been actively built where caches and even main memory are stacked vertically on top of the processor. Based on what to integrate and how to integrate them, different 3D CPU configurations have been proposed. In [105], the authors propose to keep L1 cache on the same layer with the processor while stacking all other levels of caches on top of the processors. As a result, the wire-length is shortened and the latency is reduced. Also more space will be available to place L2/L3 cache resulting in enlarged capacity. To further reduce the latency in a given level of cache, wordlines/bitlines are divided and mapped onto different active device layers [111]. To achieve more performance boost, [112] proposes to integrate the main memory on top of the processor.

When stacking a larger L2 (or other levels of cache) on top of the processor, access time will be dependent on the physical location of the requested cache line. Based on this observation, Non-Uniform Cache Architecture (NUCA) has been proposed [113] and is the most widely used model in 3D architecture. Instead of having a large and uniform L2 cache, the L2 space is divided into multiple banks, which have different access latencies depending on the location relative to the processor. See Figure 4.3, for core 1, accessing zone 1 will have the smallest latency while accessing zone 4 will have the largest latency. To improve the performance of the NUCA architecture, data migration schemes and the corresponding infrastructure have been proposed [114, 115]. The idea is to migrate data to the cache bank close to its accessing core(s).

4.3 Problem Definition

4.3.1 Attack Model

We consider computers (either single-core or multi-core) with multiple levels of memory (registers, cache, main memory,etc.). The key assumption is that there is access latency variation among different levels of memory (this is the root of non-constant time encryption and thus the root of the timing side-channel leakage). An encryption (note that we take encryption as an example of victim process here. Generally speaking, the victim process can be any process that runs on private data) process is running on this computer that takes a plaintext (ciphertext) as input and

computes the corresponding ciphertext (plaintext) with a fixed secret key $\mathbf{k}$. An adversary $\mathcal{A}$ is trying to learn the key $\mathbf{k}$. We make the following assumptions about $\mathcal{A}$.

1. $\mathcal{A}$ knows all details about the cryptographic algorithm and its implementation. Specifically, he knows the position of lookup tables in memory.

2. $\mathcal{A}$ knows how memory lines are mapped to cache lines in a conventional cache. However, the designer might hide this information from $\mathcal{A}$ by randomization.

3. $\mathcal{A}$ can feed the encryption/decryption process with chosen plaintexts (or ciphertexts) and can get the corresponding ciphertexts (or plaintexts).

4. $\mathcal{A}$ can run any program before/after or simultaneously with the encryption/decryption process.

5. $\mathcal{A}$ does not have any privileges that a super user has

Note that these assumptions are the ones generally used in the literature [3, 61, 116]. With these assumptions, the adversary $\mathcal{A}$ can apply any kinds of attacks introduced in Section 2.2 to learn the secret key $\mathbf{k}$. The major challenges for the adversary $\mathcal{A}$ are two-fold : modeling error and measurement noise. For example, in Prime+Probe attack, there are other programs and the operating system running in the background which may also evict the attacker's cache lines. These evictions are not modeled in the attack which may lead to wrong guesses.

4.3.2 Is 3D Integration a Natural Defense?

As stated in Section 4.3.1, the root cause of cache-timing side-channel attacks is the access latency variation between different levels of memory in the memory hierarchy. Since 3D integration technology can reduce access latency (thus latency variation) significantly, one natural question to ask is that if it defends against the side-channel attacks?

To explore this, we run one experiment using the Gem5 simulator [117]. The system configuration used in Gem5 is shown in Table 4.2. We use two sets of L1/L2 access latencies. For 2D cache configuration, we use 4 cycle/11 cycle as L1/L2 latency to mimic a modern Intel i7 processor [118]. For 3D cache configuration, we assume that L2 cache is partitioned into 4 dies and stacked vertically on top of the core. Using the results in [111] that 3D reduces L2 latency by 30%, we set L2 latency to 8 cycles while keeping L1 latency unchanged. We perform the one-round Prime+Probe attack proposed in [3] with an arbitrarily chosen secret key. The goal of the adversary is to learn the high nibble (highest 4 bits) of each byte i of the key, denoted by $\langle k_i \rangle$. We plot the measurement score (defined in Eqn 4.1) of $\langle k_0 \rangle$ after running the encryption with 300 different plaintexts in Figure 4.1 and Figure 4.4. Note that the ground truth is $\langle k_0 \rangle = 8$.

The maximum measurement score at both Figure 4.1 and 4.4 occurs at 8, leading the attacker to guess that $\langle k_0 \rangle = 8$. We make the following two observations from Figure 4.1 and 4.4: 1) In 3D settings, measurement score has lower signal to noise ratio. For example, in Figure 4.4, peaks at 8 and 10 have very similar

Table 4.2: Gem5 Simulator Configurations

ISA	X86
Processor type	single core, out-of-order
L1 cache	8-way, 32KB
L2 cache	8-way, 1MB
Cache line size	64 Bytes
Cache replacement policy	LRU
DRAM frequency and channels	DDR3-1600 / 1

measurement scores. Since key recovery relies solely on the adversary's ability to discover the highest peak, this adds more challenges to the adversary. In a real world setting where there are operating system and other programs running in the background, the modeling error mentioned above may cause the peak at 10 to have a larger measurement score, leading the adversary to a wrong conclusion that the high nibble of the first byte of the key is 0xA (10). This shows that 3D integration does benefit security. 2) However, $\langle k_0 \rangle$ can still be clearly spotted from the figures in both 2D and 3D cache configurations. This shows that the inherent security benefit that 3D brings is very limited.

The rest of the paper seeks other techniques that can further enhance the security of 3D integration. It should be noted that the attacks can happen at any level of cache and defending attacks at one level of cache is far from enough. Thus, the countermeasures should be comprehensive. It is also worthwhile pointing out that 3D integration technology is happening because of the low power and high performance it can bring. With our techniques, the security can be an added feature.

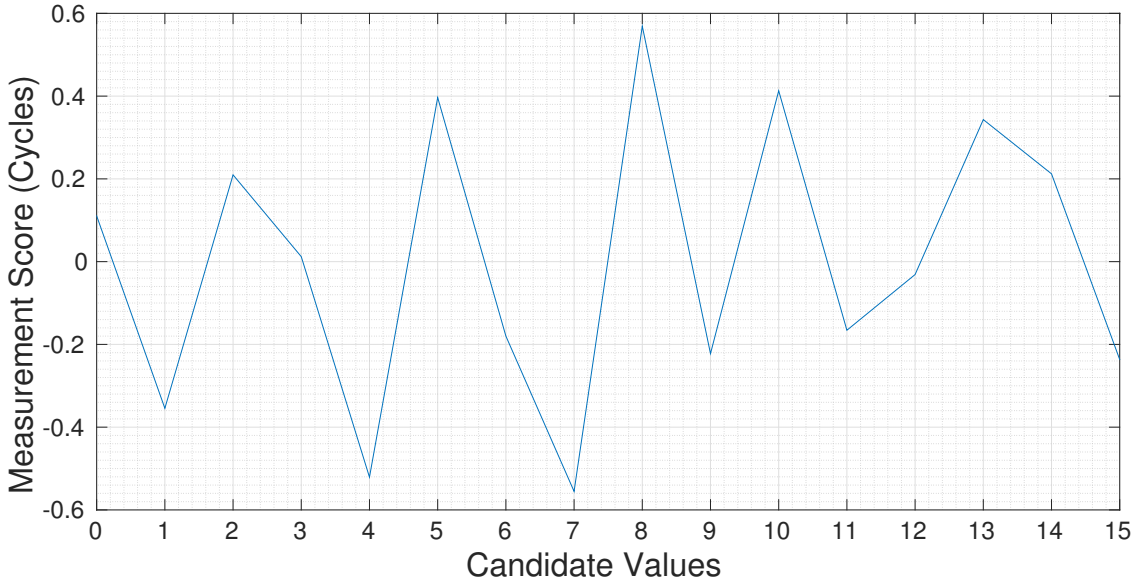


Figure 4.4: Measurement score for $\langle k_0 \rangle$ in 3D cache configurations.

Our proposed techniques, which utilizes 3D integration technology, will enable future system designers to take security into account while optimizing performance and power consumption.

4.4 Reducing Timing Side-channel Leakage

4.4.1 Side-channel Leakage Analysis

As shown in the work of Zhang et. al [55], there are three ways to ensure no side-channel leakage exists:

1. **Output Elimination**. The side-channel $\mathcal{C}$ does not produce any output.

2. **Input Ambiguity**. For any output o, the input of the channel $\mathcal{C}$ can be any input i with the same probability.

3. **Noise Domination**. For any output o, if its generation is due to the channel's inherent noise instead of the channel input, then the output is not affected by the input.

These three ways are the foundations to design countermeasures against side-channel leakage. However, completely eliminating output seems impossible. In terms of our cache-timing side-channel attacks, this requires that the attacker cannot measure any timing information from the system. Prior works suggest eliminating the instruction that enables the attacker to read the time step counter (for example, RDTSC instruction on X86 ISA) such that the attacker cannot measure the timing information. However, this is impractical since such instruction is needed in other scenarios and cannot be eliminated. Furthermore, the attacker can always use his own timing source to measure the timing information. Thus, output elimination is, though theoretically sound, not practical. In this paper, we work along the second and third approach.

To achieve input ambiguity, essentially we want to de-correlate the output from the input. In terms of our cache-timing side-channel attacks, we should ensure that timing measurements, which reflect the cache eviction or reuse patterns, are not purely (deterministically) determined by the key. This is the motivation of our random eviction, randomization-based data migration and the address permutation techniques. To achieve noise domination, we should add noise to attacker's measurement such that the signal-to-noise ratio is reduced. In terms of the cache-timing side-channel attacks, this requires a smaller difference between measurements

generated from a cache hit and a cache miss. This motivates our heterogeneous way-latency cache design and our idea of integrating more levels of cache with smaller latency difference between adjacent levels. Next, we will go over implementation details of all these designs.

4.4.2 Random Eviction Cache Design

As stated in Section 2.2, the attacker can correlate the timing information with the cache eviction (reuse) patterns. Since the eviction (reuse) is purely and deterministically decided by key-dependent data, the adversary can then discover critical information by correlating it with the timing information he measures. To de-correlate the timing information from the key-dependent data, non-determinism should be added to the system to make timing information dependent on other things besides the cryptographic key. This will make the attack harder, even impossible to learn the secret key.

The idea of random eviction, which evicts a random line from the cache at a predetermined rate, is motivated by this notion of adding nondeterminism to the system. The intuition behind how it works can be illustrated using the Prime+Probe attack introduced in Section 2.2. During the Probe phase, the attacker can measure the load time and learns whether a certain line has been evicted. If the cache eviction is deterministic, then the attacker knows whether the victim has used the cache line or not. However, if the cache eviction is non-deterministic, even if the attacker observes that a cache line has been evicted, he cannot be sure whether it is

randomly evicted by the random eviction policy or it is evicted as a result of victim accessing the cache. Thus, it prevents the attacker from learning the victim's key-dependent cache access pattern and prevents him from learning the key ultimately.

Though random eviction has been mentioned in several papers [55, 119], no implementation details and experimental results have been given (especially in the context of 3D integration). The main drawback of this technique is the associated performance overhead, since useful data may be evicted from the cache, causing high miss rate. However, we argue that the overhead can be greatly reduced with the use of 3D integration. We use the data from [111] to help illustrate this. [111] showed that the L2 access latency can be reduced by 30% with 3D integration. Therefore, if the data is randomly evicted from L1 cache, the resulting penalty of accessing L2 cache is reduced by 30%, which is a huge improvement. Furthermore, the penalty would be fairly small compared with the huge performance gain that 3D brings. Thus, we argue that 3D integration enables the random eviction policy to take place. Next, we investigate some implementation details which have not been given in the prior literature.

Note that in a modern multiple-core computer, evicting a line from the cache cannot be done by simply setting the invalid bit of the cache to 1 and writing back the data. Coherence policies have to be enforced so that when one share of data gets invalidated and written back to the memory, the other copies get updated. The proposed random eviction cache architecture is shown in Figure 4.5.

Eviction Interval Register is used to control the interval between two random evictions. For example, if it is five, then every 5 cycles, a random line is evicted

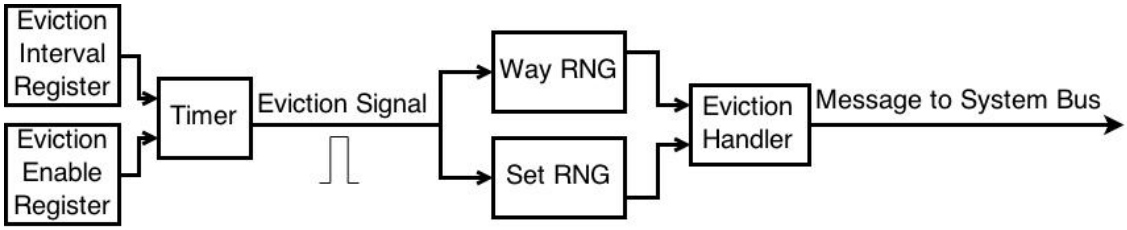


Figure 4.5: Block Diagram of Random Eviction Policy

from the cache. Intuitively, the smaller this number is, the larger de-correlation can be achieved and thus more robust the system is against timing side-channel attacks. However, smaller eviction interval also leads to larger downgrade in performance. Thus, this number governs the trade-off between security and performance, and has to be decided smartly. The trade-off between performance and security will be shown later in Section 4.5. The Eviction Enable Register is used to enable/disable the random eviction policy. According to the values of these two registers, a timer is used to generate proper eviction signals. Then two random number generators (RNGs) are used to generate random way and set number respectively. These two numbers are fed to eviction handler to handle the eviction (and possible write-back) and generate proper messages to be broadcast to the system bus to enforce cache coherence. The detailed implementation of eviction handler is dependent on the coherence policy. We give an example implementation for MOESI protocol in Table 4.3 which is a coherence protocol commonly used in modern processors.

The random eviction policy can be implemented with/without operating system's assistance. Without the operating system's assistance, the Eviction Interval Register and Eviction Enable Register are hard-coded and the random eviction

Table 4.3: The actions to perform and messages to broadcast to the system bus by eviction handler depending on the cache line state. The cache line is marked as I state after performing actions and broadcasting messages.

Cache State	Actions	Messages
M	Set invalid bit and write back the cached data	None
O	Set invalid bit and write back the cached data	Invalidate lines
E	Set invalid bit	None
S	Set invalid bit	None
I	None	None

policy is always happening. With the operating system's assistance however, the operating system is in charge of filling these two registers. When the system is running non-security-critical programs, the random eviction policy can be turned off for maximizing performance. When the system is running security-critical programs, the random eviction policy should be enabled and the random interval should be set according to the level of security needed. These two methods have their own advantages/disadvantages and should be chosen at the designer's discretion. With operating system's assistance, the random eviction policy can be turned on only when needed and thus maximized performance can be achieved. However, this makes the attack's job simpler. He only needs to hack the operating system to set the Eviction Enable Handler to 0 to bypass the random eviction policy. Without operating system's assistance, the attacker has no way to turn off this policy but this downgrades the system's performance when executing programs that are not security-critical.

4.4.3 Heterogeneous Way-Latency Design

Our second design aims to add noise and reduce the signal-to-noise ratio of the attacker's measurement. Traditionally, accessing the data in each way of the same cache should have the same access latency. We propose a cache design that has different accessing latencies across different ways and the latencies are determined as follows. Suppose in a conventional N_{way}-way set associative cache, the latency for accessing this level and next level of cache is t_{up} and t_{down} respectively. Then the latency for accessing each way i, $i = 0, 1, \ldots, N_{way} - 1$ of our heterogeneous way-latency cache, denoted by t_i, is determined by:

$$t_i = t_{up} + \frac{t_{down} - t_{up}}{N_{way}} \times i \tag{4.4}$$

Rounding to the nearest integer is performed, if needed. For instance, suppose in a 8-way set associative conventional cache, the latency for accessing this and next level of cache is 4/8 respectively. Then in our cache design, accessing data in way-0 through way-7 will have a latency of 4, 4, 5, 5, 6, 6, 7, 7 respectively. The intuition behind how this design can defend against side-channel attacks is illustrated as follows.

Take Prime+Probe attack as an example. The adversary eventually will measure the time to reload his data and judges whether his data has been evicted based on that time. Suppose the cache hit/miss latency is 4/8 cycles respectively and the cache is 8-way set associative. In a conventional cache, the time to reload the attacker's data in a set if the victim does not evict any of his data is 4*8=32 cycles.

If the victim has evicted one cache line from this set, the time for the attacker to reload is 4*7+8*1=36 cycles. There is a 4-cycle difference. However, in our cache design where way-0 through way-7 have a latency of 4, 4, 5, 5, 6, 6, 7, 7 cycles respectively, the difference can be 1-4 cycles depending on which way gets evicted. The signal-to-noise ratio can be reduced by as much as 75%. The 1 or 2 cycle of difference can be hard to measure given the presence of background noise. Similarly, this cache design also adds noise to internal interference based attacks.

Using voltage scaling, the technique can be easily implemented by slowing down the sense amplifiers of the bit lines of different ways. It is important to note that though this method adds randomness to the attacker's measurement, it also downgrades the performance. However, with the 3D integration technology which reduces the latency difference between two levels of cache, the performance overhead is much smaller than in a 2D configuration. Thus, 3D integration enables this defense mechanism to take place.

The heterogeneous way-latency design can also implemented purely in hardware (where the voltage scaling is permanent) or with the assistance from the operating system (where the voltage scaling is dynamically determined by the operating system). The pros and cons of these two methods are similar to those in the random eviction policy.

4.4.4 Components to Integrate

As stated above, attacks may happen at any level of cache. A secure system should defend against all of them. Since the root cause of all these attacks is the huge latency difference between different levels of the cache and main memory (see Section 2.2), we argue that the following claim is true:

Claim 1 *Integrating more levels of cache so that there is a more gradual change in latency from cores to main memory is beneficial for security.*

We illustrate this using the following example. Consider two options when designing a 3D CPU. We can either stack large L2/L3 caches on top of the processors (configuration 1) or we can choose to sacrifice some L2/L3 capacity and use the saved space for a DRAM L4 cache (configuration 2). Prime+Probe attacks on L1/L2 will have the same results in both configurations since the latency difference between L1/L2 and L2/L3 are the same. Note that cache capacity does not matter because the attacker will only be using a very small portion of the cache. However, attacks on L3 will have dramatically different results. Assume that L3/L4 cache and the main memory have a latency of t_3, t_4, t_m respectively. Assume in the attack, one cache line containing attacker's data in L3 is evicted by the victim process. Then the evicted line has to be fetched from the main memory in configuration 1 and from L4 in configuration 2. As a result, in configuration 1, there will be a $t_m - t_3$ cycle-difference in the attacker's measurements depending on whether his data gets evicted by the victim process or not. That difference is reduced to $t_m - t_4$ cycles in configuration 2 (note that $t_4 > t_3$). Consequently, configuration 2 has a smaller

timing leakage, which demonstrates our Claim 1.

For time-driven attacks, more levels of cache is also beneficial. This is because in these attacks, the attacker tries to correlate the secret information (such as the cryptographic key) with the overall execution time, which depends on the number of cache hits in different levels of cache. More levels of cache will add more variables to the system, which will significantly complicate the correlation process. Note that fundamentally this is true to both 2D and 3D designs. However, 3D CPUs are more relevant because they have more space to allow additional caches to be added.

4.4.5 Data Migration Scheme

As stated in Section 4.2.3, L2 cache and below will most likely be NUCA in 3D CPUs. Though in some 2D designs, NUCA has already been applied, we argue that data migration scheme (DMS) is more relevant in 3D because of locality and network on-chip. Therefore, we consider the impact of DMS on security in this subsection. We use Figure 4.3 to illustrate NUCA. L2 cache is stacked on top of four cores. It is divided into 16 banks. The organization of 16 ways in a particular cache set is shown in Figure 4.3. Suppose the latency of core 1 accessing data in zone 1/2/3/4 is t_1, t_2, t_3, t_4 respectively. Due to locality, $t_4 > t_3 > t_2 > t_1$. Without data migration, the data block often accessed by core 1 may be located at zone 4, which is very inefficient. To optimize the performance, researchers have proposed the following performance-oriented data migration scheme : periodically check and migrate the data to the zone nearest to its accessing core(s) [115]. However, its

security implication is unknown. We consider a randomization-based DMS which periodically migrates a data block to a random zone. We argue that the following claim is true:

Claim 2 *A randomization-based DMS is more beneficial to security than a performance-oriented DMS*

To show this, we compare the randomization-based DMS with above performance-oriented DMS. We still use the setting in Figure 4.3. In time-driven attacks, the attacker will try to correlate the secret information (such as the cryptographic key) with the overall execution time. If a performance-oriented DMS is applied, then zone 1 will host data that are recently accessed by core1 and zone 4 will host data that are least recently used by core1. As a result, when a cache miss happens, most likely a data block from zone 4 will be replaced. This fixed pattern makes the attacker's correlation easier. On the contrary, if a randomization-based DMS is applied, data block in zone 1/2/3/4 may be replaced with equal probability. This adds much more randomness into the overall execution time, making the attack much harder.

The randomization-based DMS also adds security against access-driven (such as Prime+Probe) attacks. For illustration purposes, we consider attacks on L2 cache. Let t_{l3} denote the average L3 hit latency. If one cache line containing attacker's data in L2 is evicted by the victim process in the attack, in a performance-oriented DMS, the victim's data block will be migrated to zone 1 (because it has been recently accessed). As a result, there will be a $t_{l3} - t_1$ cycle-difference in the attacker's measurements between the case where the attacker's data was evicted

by the victim process and the one where his data was not evicted. On the other hand, if randomization-based DMS is applied, the victim's data block can reside in zone 1/2/3/4 with equal probability. Therefore, the expected difference between two cases is $t_{l3} - \frac{t_1+t_2+t_3+t_4}{4}$. Since $\frac{t_1+t_2+t_3+t_4}{4} > t_1$, the expected difference is smaller with randomization-based DMS applied. Therefore, we conclude that a randomization-based DMS results in smaller timing leakage, which demonstrates Claim 2.

Note that we do not apply DMS to L1 cache because: 1) L1 cache is on-core where network-on-chip is not applicable. Data cannot travel efficiently. 2) L1 cache is sensitive to latency and the traffic overhead of migrating data blocks is too large for L1. Therefore, we only apply randomization-based DMS to L2 cache and below. Consequently, it does not defend against Prime+Probe attacks on L1 cache.

4.4.6 Address Permutation

Section 4.4.4 and 4.4.5 have addressed two fundamental problems unique to 3D CPU design. However, attacks on L1 cache (both instruction and data) and branch target buffer (BTB) are still not mitigated. In this section, we propose the address permutation technique to mitigate attacks on all levels/types of cache, including L1 cache and BTB.

The idea of address permutation is to **permute the address lines randomly** (not known to the attacker). We illustrate this using the following example. Assume the attacker is performing Prime+Probe attack on a 4-way set associative cache to learn whether cache set 1 was accessed by the victim process or not. As-

sume the cache has $2^3 = 8$ sets and each cache block has $2^2 = 4$ bytes. In this setting, the cache set number is obtained by extracting bit4-bit2 in the address line (shown in Figure 4.6 as shaded box). In the Prime step in the attack (see Section 2.2), the attacker will fill the cache set 1 by loading the data in four addresses shown in the left hand side of Figure 4.6. Note that these four addresses are all mapped to cache set 1. Next consider the case where permutation of the address line $\{b7, b6, b5, b4, b3, b2, b1, b0\} \rightarrow \{b5, b3, b7, b4, b2, b6, b1, b0\}$ is applied (but the attacker is not aware of this). The permuted four addresses are shown in the right hand side of Figure 4.6. The attacker still loads data in these four addresses hoping to fill cache set 1 with his data. However, in this case, he is actually loading data to cache set 2, 2, 3, 3 respectively, none of which are related to cache set 1. Therefore, the attacker cannot gain any information about whether cache set 1 was accessed or not.

The implementation diagram is given in Figure 4.7. The permutation network can be implemented efficiently using Benes network [120] or Butterfly network [121]. Note that for performance reasons, only the block address is permuted. This means that if a cache block is $2^6 = 64$ bytes, the lowest 6 bits are not permuted. CPU can change the permutation by changing the value in the permutation control register. The register can be changed at a given interval or before running security-critical applications. Every time the permutation control register is changed, all dirty cache lines need to be written back to the memory. However, we will show next that the permutation control register does not need to be changed very often, thus the write-back overhead can be amortized. In modern processors, L1 cache is virtually

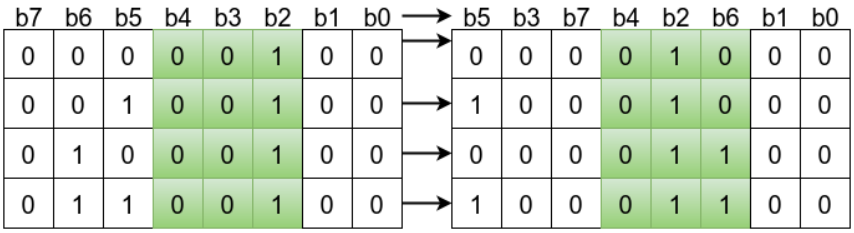


Figure 4.6: Address Permutation Illustration.

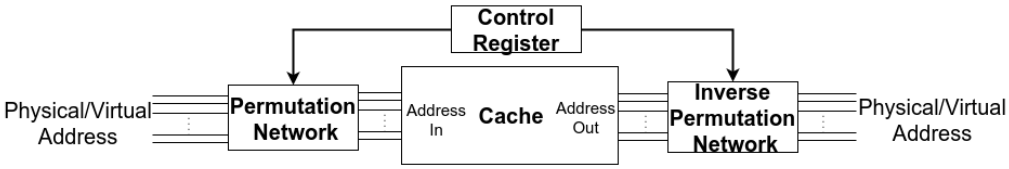


Figure 4.7: Address Permutation Block Diagram.

indexed, physically tagged while all other caches are physically indexed, physically tagged. Thus the permuted virtual address is only connected to L1 cache while permuted physical address goes to all caches. The tag is extracted from the permuted physical address in all caches. When write-back happens, original physical address will be computed through the inverse permutation network.

Note that the above technique is only effective against access-driven attacks. It may be applied to any level/type of cache such as instruction cache or branch target buffer. The security of this technique depends on the fact that the attacker does not know what the permutation is. This is ensured by making permutation control register access a privileged instruction. However, the attacker may still learn the permutation through brute-force attack. Note that the attacker only needs to determine the permutation relevant to cache set extraction in all levels of cache rather than the full permutation. Consider the following setting similar to a modern

64-bit computer. The cache block is $2^6 = 32$ bytes and the maximum number of cache sets in all caches is $2^{13} = 8192$. We will illustrate the brute-force attack as follows.

The attacker's job is to figure out which 13 bits out of the 64-6=58 bits are used as set index. The attacker constructs C_{58}^{13} hypotheses, each corresponding to one possible address mapping. Then for each address mapping, the attacker finds 17 addresses that map to the same cache set under this address mapping. The attacker then primes the cache using the first 16 addresses, visit the 17^{th} address, and then reload data in the first 16 addresses. The attacker measures the time in the prime step and probe step respectively. The time difference between these two time steps is the signature for that hypotheses. The true hypothesis is the one with the largest signature. We simulate the above steps on our computer. The attacker is able to obtain the signature values for 7×10^5 hypotheses each second. Then getting the signature for all hypotheses would take roughly 52 days. In other words, the attacker needs roughly 52 days to successfully guess the correct address mapping. Therefore, the control register does not need to be changed often so the write-back overhead can be neglected.

We note that the idea of randomizing memory-cache address mapping has been proposed in [77]. However, there are some salient differences between our method and theirs. Their work uses a level of indirection and involves the lookup of the index bits in the ReMapping Table (RMT) which may be very inefficient. On the contrary, our method manipulates directly on the address mapping and incurs very small performance overhead (this can be verified by our experiments

in Section 4.5). Moreover, it is unknown whether their work can extend to BTB since they fail to show the performance and security evaluations on the BTB. Our countermeasures, however, are very effective against attacks on BTB with very small overhead (see Section 4.5). We also note that modern Intel processors divide LLC into four slices and use an undocumented hash map (which is the xor of some bits of the physical address) to determine which slice a memory block resides in. Though this essentially is another form of memory-cache address randomization, the level of security it achieves is too weak. In a 32-bit processor, only the highest 15 bits of the physical address are involved in the hash map. This means an attacker only needs to try $2^{15} = 32768$ combinations before he can successfully recover the hash map (note that with our address permutation technique, the attacker needs to try out 6.48×10^{16} permutations). Actually, there already have been some works that successfully reverse-engineer the hash map [56, 122].

4.5 Evaluations

In this section, we will evaluate the performance and security implications of our proposed countermeasures. We first introduce the metrics and simulation infrastructure we will be using. Then we explain in detail the experiments we conduct and discuss the results. Note that the experimental evaluation of the countermeasures proposed in Section 4.4.2 and 4.4.3 appears in [43]. We will only evaluate the rest of the countermeasures and all countermeasures combined in this paper.

4.5.1 Metrics and Simulation Infrastructure

Metrics. For performance evaluation, we record the instruction per cycle (IPC) after running each benchmark for 10 billion instructions. This represents the number of instructions a system can run in a single cycle. The larger this value is, the better the performance is. For security evaluation, we calculate the side-channel vulnerability factor (SVF). More details about SVF can be found in [119]. Basically, SVF is a value between 0-1 that measures the correlation between the attacker's measurements and the ground truth. Thus, lower SVF translates to smaller side-channel leakage. Though [123] proposed another metric based on SVF, we feel that SVF is more suitable for our situation for accurate system side-channel leakage characterization since [123] is based on a binary observation trace while SVF works on observation traces that take any numerical values. Therefore, for our evaluation purposes, we will use SVF as security metric. The larger SVF a system has, the less secure it is.

Simulation Framework. We implement all the proposed techniques on the Gem5 simulator [117], which is a cycle-accurate, modular simulator widely used for computer architecture research. We simulate a two-core X86 CPU with different memory system configurations. For illustration purposes, we show the detailed configuration for a 2D baseline system in Table 4.4. In different experiments, we may add more levels of cache and change cache size/latency to simulate 3D CPUs. The parameters will be introduced later in this section. For performance evaluations, we run 10 PARSEC benchmarks [124] on each system and record the corresponding

Table 4.4: Gem5 Configuration of a 2D Baseline System

ISA	X86
Processor type	2 cores, out-of-order
L1 cache	8-way, 32 KB, 4-cycle latency
L2 cache	8-way, 1MB, 11-cycle latency
L3 cache	16-way, 3MB, 35-cycle latency
Cache line size	64 Bytes
Cache replacement policy	LRU
Main Memory	Gem5's SimpleMemory, 200-cycle latency

IPC. This shows the performance overhead of our countermeasures on a general-purpose computer. For security evaluations, we simulate several attacks on Gem5 and calculate SVF for each attack. The attacks we simulated will be introduced shortly.

Attacks performed. To evaluate the security implications, we simulate the following four widely-applied attacks. 1) We perform Prime+Probe attack on L1 cache to attack an AES algorithm, following the steps in [3]. 2) We follow steps in [56] to perform Prime+Probe attack on the last level of cache (LLC) to deduce the cache set used by the victim process. 3) We perform a time-driven attack introduced in [2] to attack an AES algorithm. 4) We perform the branch prediction attack (BPA) introduced in [5] to deduce the key used in the victim process. The victim process would either take or not take a branch based on each bit of a 32-bit key.

We perform five experiments to evaluate the security and performance of our

Table 4.5: Different systems we have simulated.

Systems Name	Eviction Interval	Heterogeneous Way Latency Used
2D_base	No Evictions	No
2D_5N	5 Cycles	No
2D_5Y	5 Cycles	Yes
3D_ins	No Evictions	No
3D_5N	5 Cycles	No
3D_2N	2 Cycles	No
3D_5Y	5 Cycles	Yes

proposed techniques. The first four experiments evaluate each of the proposed techniques. The fifth experiment evaluates all the techniques combined.

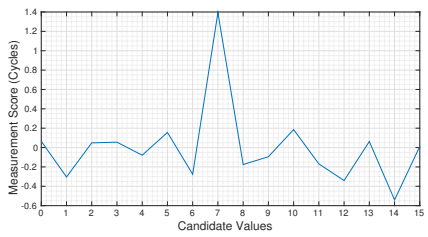
In the first experiment, we evaluate the security and performance implications of random eviction technique (RET) and heterogeneous-way latency (HWT) technique. To be more specific, we explore the performance and security trade-off under different parameters. Seven different systems are simulated and their parameters are listed in Table 4.5. In this experiment, the L1 cache is private on-core while the L2 cache is shared among different cores. The access latency for L1/L2 cache in the 2D design is 4/11 cycles respectively. We assume that using 3D integration, L2 cache is partitioned into 4 layers, connected by TSVs, and is similar to the one shown in Figure 4.2. This fine-grained partition reduces the interconnect length and thus the access latency for L2. Using the data in [111], we set the L1/L2 latency to 4/8 cycles respectively. Similarly, we assume that the main memory (DRAM) is also integrated on-chip using 3D integration, we cut the DRAM controller latency

in half [125]. Other configurations are exactly the same as in 2D system. The corresponding system is called the 3D insecure system (3D_ins).

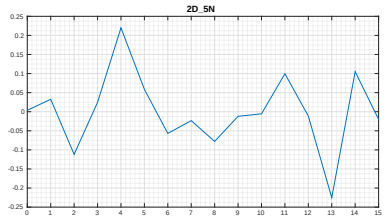
We evaluate security under two different attacks, Prime+Probe attack and Bernstein's attack. In Prime+Probe attack, we follow the one-round attack steps in [3] with one exception: we do not use the "pointer chasing" technique since we find it unnecessary. The goal is to extract the high nibble (highest four bits), denoted by $\langle\cdot\rangle$, of each byte of the 16-byte key. The key is chosen arbitrarily. We plot the measurement score of the 16-th byte on each simulated systems (see Table 4.5) after 1000 encryptions in Figure 4.8.

A visual inspection of these measurement scores shows that in 2D_base and 3D_ins systems, the measurement score of 16^{th} byte takes its largest value at 7, which leads the attacker to the correct conclusion that $\langle k_{15}\rangle = 7$. However, $\langle k_{15}\rangle$ cannot be correctly deduced from any other systems. The average SVF is reported in Table 4.6. From Table 4.6, we can make several observations: 1) The 2D_base has a slightly larger SVF than 3D_ins system. This verifies our hypothesis that with a reduced latency, 3D integration offers inherent security benefits (but very limited). 2) The larger eviction frequency a system has, the larger security improvement it can achieve (3D_2N has a smaller SVF than 3D_5N). 3) The SVF goes down significantly as we add random eviction and heterogeneous way-latency mechanisms to the system. 4) 3D_5Y has a significantly smaller SVF than 3D_2N, which means security is maximized when both defense mechanisms work together.

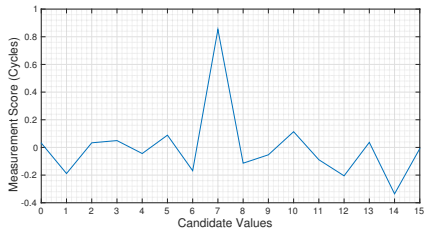
In Bernstein's attack, we follow the steps in [2] with the packet size set to 16 bytes and 2^{22} samples (The original attack uses 2^{27} samples. However, taking 2^{27}



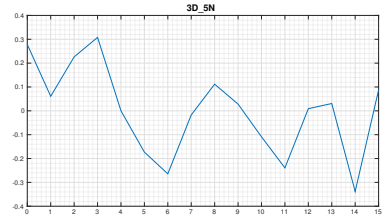
(a) 2D_base



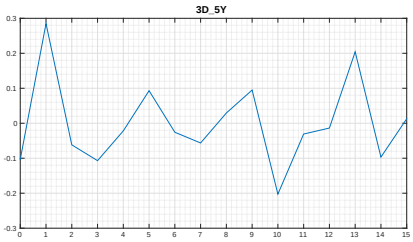
(b) 2D_5N



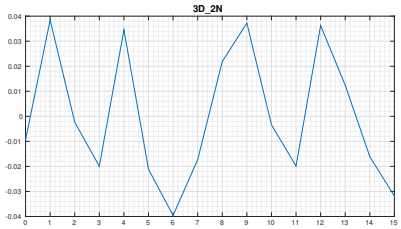
(c) 3D_ins



(d) 3D_5N



(e) 3D_5Y



(f) 3D_2N

Figure 4.8: Measurement score for 16^{th} byte of the key in different systems. X-axis is the candidate value, Y-axis is the measurement score.

samples on the simulator would take more than 25 days, which is too long. Moreover, it is enough to show the efficacy of our countermeasures using 2^{22} samples). The average SVF is shown in Table 4.6. The discussion for the Prime+Probe attack also applies here. One thing to note is that the SVF improvement is not as large as that in Prime+Probe attack. This implies that Bernstein's attack, or internal interference attacks in general, is much harder to defend than external interference

Table 4.6: The average SVF in Prime+Probe attack and Bernstein's attack for different systems

System	Prime+Probe	Berstein
2D_base	0.8743	0.1071
2D_5N	0.1085	0.0538
2D_5Y	0.0723	0.0369
3D_ins	0.8628	0.0929
3D_5N	0.0668	0.0511
3D_2N	0.0589	0.0498
3D_5Y	0.0247	0.0348

attacks as stated in Section 2.2. Nevertheless, our proposed methods can still cut the SVF by more than two thirds.

In performance evaluation, we evaluate the performance of our proposed mechanisms on 7 PARSEC benchmarks [126]. The normalized instruction per cycle (IPC) on general purpose programs as well as on AES encryption of 10 thousand 16-byte plaintexts is recorded. We report the performance gain (IPC improvement) over 2D_base for each benchmark on each system. The average performance gain on all Parsec benchmarks are also reported. The results are shown in Figure 4.9.

The following observations can be made from the results: 1) On average, the 3D_ins has 20.62% higher IPC than 2D_base. This reflects the potential huge performance benefit by transition from 2D to 3D. 2) If we compare 3D_5Y with 3D_ins and compare 2D_5Y with 2D_base, on average, 3D_5Y has 5.58% performance overhead while 2D_5Y has 10.73% overhead. The performance overhead is smaller

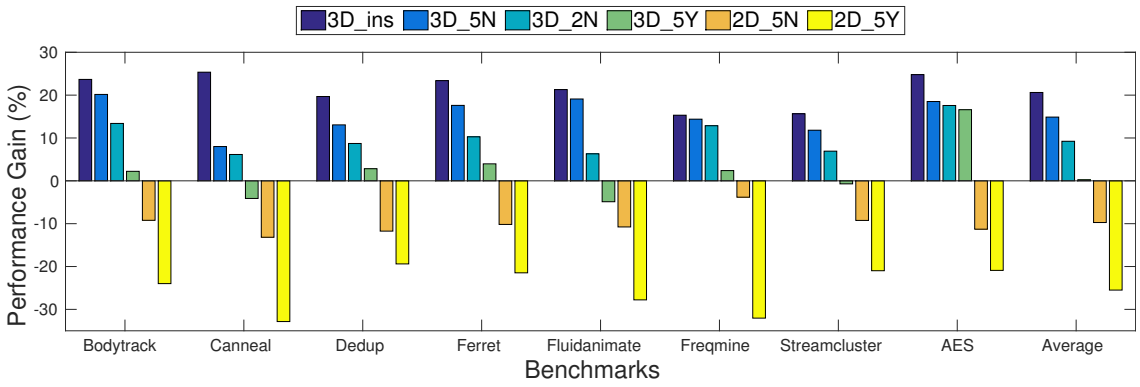


Figure 4.9: Performance Gain of RET+HWL Over 2D Baseline System.

in 3D since the penalty for a cache miss is smaller, as discussed in Section 4.4.2. 3) As the eviction frequency goes up, the performance overhead increases. This is intuitively right because higher eviction frequency leads to more data elimination and downgrades the performance. 4) On all benchmarks, 3D_5Y has around 10% worse performance than 3D_2N, however, Table 4.6 shows that the improvement in security is huge (thus 3D_5Y should be favored over 3D_2N). In other words, our two countermeasures should work together to obtain the maximum security over performance yield. 5) It is noteworthy that on most countermeasures, our strongest countermeasure (3D_5Y) still performs better or equally good as 2D_base (on average, 0.24% gain). This leverages the huge improvement performance 3D brings. Furthermore, with operating system's assistance where countermeasures will be applied only when executing security-critical program, a secure system (3D_5Y) still has 16.60% performance gain over 2D baseline system (on AES). In a word, the performance loss due to our proposed mechanisms would be fairly small compared to the huge gains obtained by going to 3D. Therefore, we conclude that 3D integration

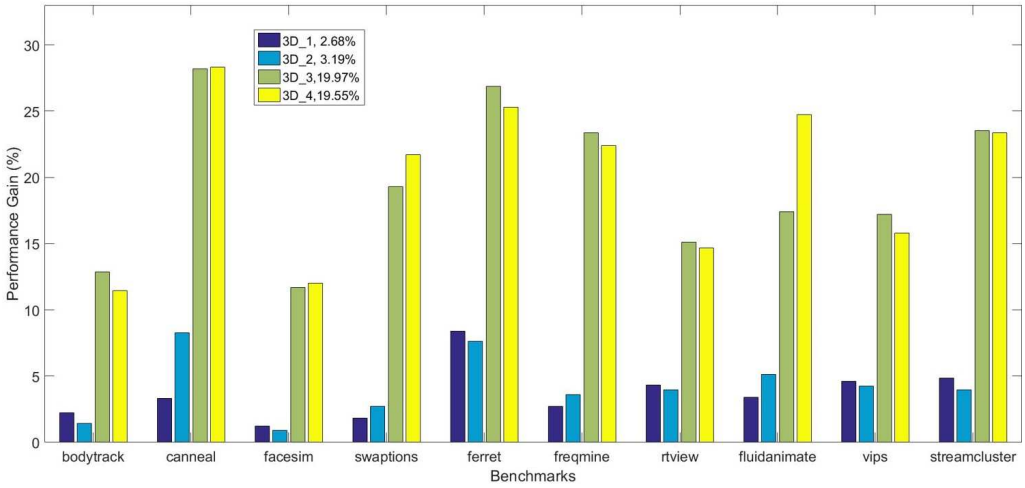


Figure 4.10: Performance Gain of 3D_1, 3D_2, 3D_3, 3D_4 Over the 2D_base System.

enables random eviction and heterogeneous way latency mechanisms, which would have been impossible in a 2D context, to take place.

The second experiment verifies Claim 1 in Section 4.4.4, that is more levels of cache with smaller latency difference between two adjacent levels is beneficial to security. Five systems are simulated, shown in Figure 4.7. We introduce each of them in more detail:

- 2D_base represents a 2D system. The L1, L2, L3 latency is set to 4, 11, 35 cycles respectively, which mimics a modern Intel processor [118]. The size of L1, L2, and L3 caches also resembles the one used in a modern processor.

- 3D_1 represents a 3D processor with main memory off-chip. Compared with 2D_base, we add more caches (1M of L2, 3M of L3 and 16M of L4, all SRAM) to capture the fact that 3D processors have more available resources. The latency of L1∼L3 is also reduced to 4, 8 and 18 cycles due to shorter wire-length that 3D integration enables. The latency of L4 is set to 40 cycles.

- 3D_2 also represents a 3D processor with main memory off-chip. Compared with 3D_1, we reduce the size of L2/L3 SRAM caches and use that space for an additional L5 DRAM cache. Note that since the area of 1M DRAM and 8M SRAM is roughly the same, 3D_1 and 3D_2 have roughly the same total area. L5 latency is set to 90 cycles. All other latencies are the same as 3D_1.

- 3D_3 represents a 3D processor with main memory on-chip. Compared with 3D_1, we take away L4 SRAM cache and use that space for on-chip main memory. Still, we keep the total area the same. The on-chip main memory latency is set to 100 cycles. All other latencies are the same as 3D_1.

- 3D_4 also represents a 3D processor with main memory on-chip. Compared with 3D_3, we reduce the size of L2/L3 caches and use that space for L4 cache. All latencies are the same as 3D_3.

For this experiment, we assume that NUCA is not applied which means a uniform access latency is used for each level of cache. We evaluate the performance of each system on 10 PARSEC benchmarks. As shown in Table 4.8, our first technique does not defend against branch prediction attacks and prime+probe attacks on L1, therefore, we only evaluate security on prime+probe attacks on LLC and time-driven attacks.

The performance gain over 2D_base is shown in Figure 4.10. The geometric mean of the performance gain for each system in shown in the legend. We also calculate SVF for each attack on each system and plot the SVF reduction over 2D_base in Figure 4.11.

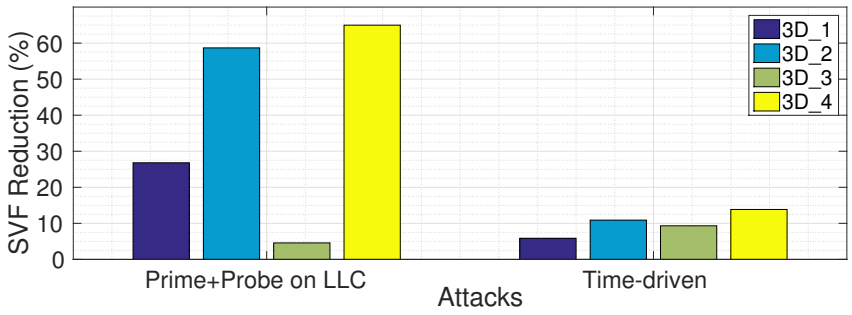


Figure 4.11: SVF Reduction of 3D_1, 3D_2, 3D_3, 3D_4 Over the 2D_base System.

Table 4.7: Systems Simulated

Systems	L1 Size	L2 Size	L3 Size	L4 Size	L5 Size	Main Memory (On-Chip) Size
2D_base	32K	1M	3M	-	-	-
3D_1	32K	2M	6M	16M	-	-
3D_2	32K	1M	3M	16M	32M	-
3D_3	32K	2M	6M	-	-	128M
3D_4	32K	1M	3M	4M	-	128M

Comparing 3D_1 with 3D_2 and 3D_3 with 3D_4 in Figure 4.11 and 4.10, we can see that having an extra level of cache (at the cost of reducing the capacity of high-level caches) has negligible overhead in performance but is extremely beneficial to security. The security gain is especially large for Prime+Probe attack on LLC, since an extra level of cache can remarkably reduce the latency difference between LLC and main memory.

In the third experiment, we compare the randomization-based data migration scheme with performance-oriented data migration scheme (see Section 4.4.5). The details of these migration schemes can be found in Section 4.4.5. We assume that

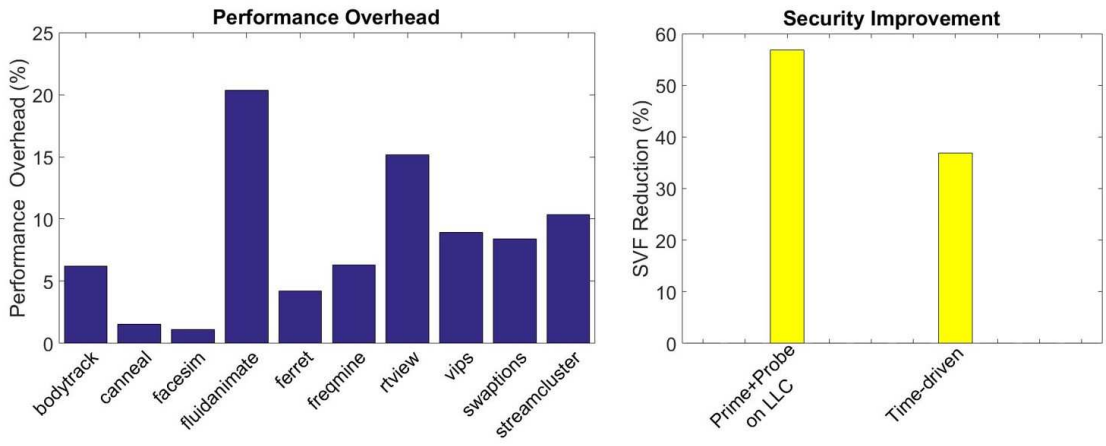


Figure 4.12: Evaluation of Randomization-based DMS with Respect to Performance-oriented DMS.

they are invoked every 100 clock cycles. We use 3D_3 in Figure 4.7 as our memory system and implement two migration schemes on top of it. Note that if other memory systems are chosen, we will get similar results. We choose 3D_3 as our memory system without loss of generalization. We compare the randomization-based data migration scheme (DMS) with respect to performance-oriented DMS. In this experiment, NUCA is applied to L2 and L3. The L2 and L3 layers are divided into four zones (see Figure 4.3). We assume that for core 1, accessing zone 1∼zone 4 will have an additional 0, 2, 4, 8-cycle hop latency on top of the latency values used in the first experiment. The latency for core 2 can be determined similarly.

The performance (IPC) overhead and SVF reduction of randomization-based data migration scheme with respect to performance-oriented data migration scheme are plotted in Figure 4.12. Note that as shown in Table 4.8, randomization-based

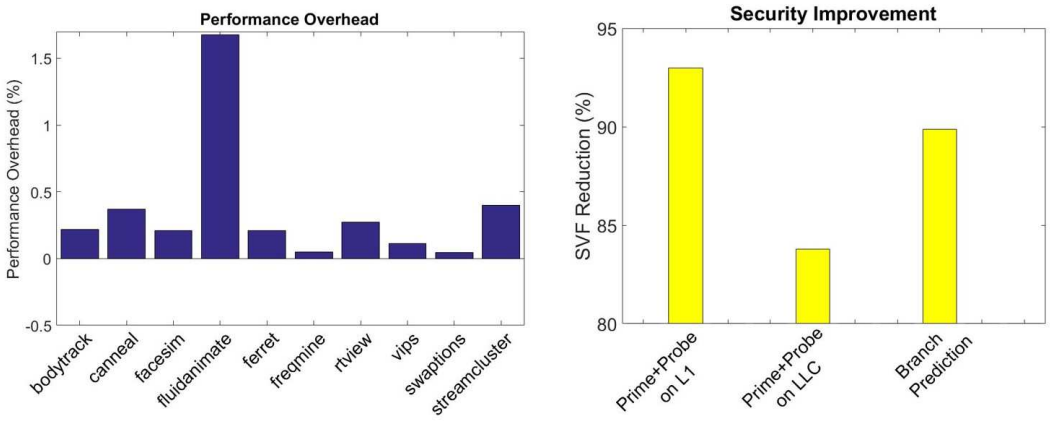


Figure 4.13: Evaluation of Address Permutation Technique.

DMS only benefits Prime+Probe attacks on LLC and time-driven attacks, therefore we only calculate SVF data on these two attacks. On average, randomization-based data migration scheme has 5.82% performance overhead, but can achieve over 35% SVF reduction in two attacks.

In the fourth experiment, we implement the address permutation technique introduced in Section 4.4.6 to all levels of cache (including the branch target buffer). We use 3D_3 in Figure 4.7 as our memory system and apply address permutation to it. Note that if other memory systems are chosen, we will get similar results. We choose 3D_3 as our memory system without loss of generalization. In this experiment, NUCA is not applied. As stated in Section 4.4.6, address permutation is not effective against time-driven attacks. Therefore, we evaluate security against all attacks but time-drive attack introduced in Section 4.5.1. We randomly pick 20 permutations and calculate the performance overhead and SVF reduction of each permutation. The averaged performance overhead and SVF reduction (with respect

to a system without address permutation implemented) are reported in Figure 4.13. The results show that with at most 1.4% performance overhead, the address permutation technique can reduce SVF by over 85%.

In the last experiment, we compare random eviction technique (RET) and heterogeneous way-latency (HWL) with all other techniques proposed in Section 4.4.4, 4.4.5 and 4.4.6. We implement randomization-based DMS and address permutation on system 3D_4 shown in Figure 4.7. We also implement RET and HWL on system 3D_3. Note that we choose 3D_3 instead of 3D_4 because 3D_4 shows the effectiveness of redistributing cache levels. Then we implement randomization-based DMS, address permutation, RET and HWL on top of system 3D_4, which represents the system with all the strongest countermeasures proposed in both this work. We evaluate the performance by obtaining the IPC values running 6 different PARSEC benchmarks. We evaluate the security by simulating four attacks introduced in Section 4.5.1 and calculate the SVF. We compare all the results with respect to a 2D baseline system 2D_base. The performance gain and SVF reduction over a 2D_base baseline system are plotted in Figure 4.14. The geometric mean of performance gain is shown in the legend. We also compare the performance of our strongest countermeasures (all proposed countermeasures on top of 3D_4) with a 3D baseline system (performance-oriented DMS+3D_3). On average, the performance downgrade is 6.86%.

Figure 4.14 shows that on most attacks, the security improvement of RET+HWL and other techniques is comparable. However, other techniques have far less performance overhead. One exception is that RET+HWL is more effective on time-driven

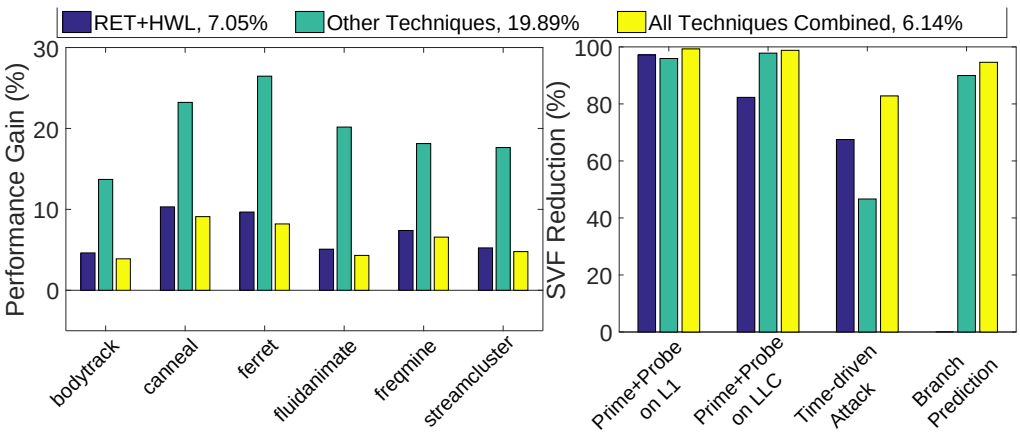


Figure 4.14: Comparison between different techniques proposed in this thesis.

attacks. Therefore, to achieve the strongest security, we propose to ombine all techniques proposed in this work. The security evaluation shows that the combined system achieves the highest SVF reduction at the cost of some additional overhead.

To demonstrate that our proposed countermeasures achieve satisfactory SVF reduction (and thus side-channel leakage reduction), we plot the attacker's measurements in Prime+Probe attacks on L1/LLC, in time-driven attacks and in branch prediction attacks on a 2D baseline system 2D_base and on 3D_4 system with all our proposed countermeasures applied.

We plot the measurement scores for 16^{th} key byte in Prime+Probe attacks on L1 in Figure 4.15. More details on the attack can be found in [3]. Basically the measurement scores are related to the reload time in the third step of the Prime+Probe attack introduced in Section 2.2. The attacker will deduce the highest 4 bits of each byte of the key based on the measurement scores. He will guess that the candidate value with the largest measurement score is the right one. In our settings, the correct value is 7. We can see that in a 2D_base system, the attacker can correctly

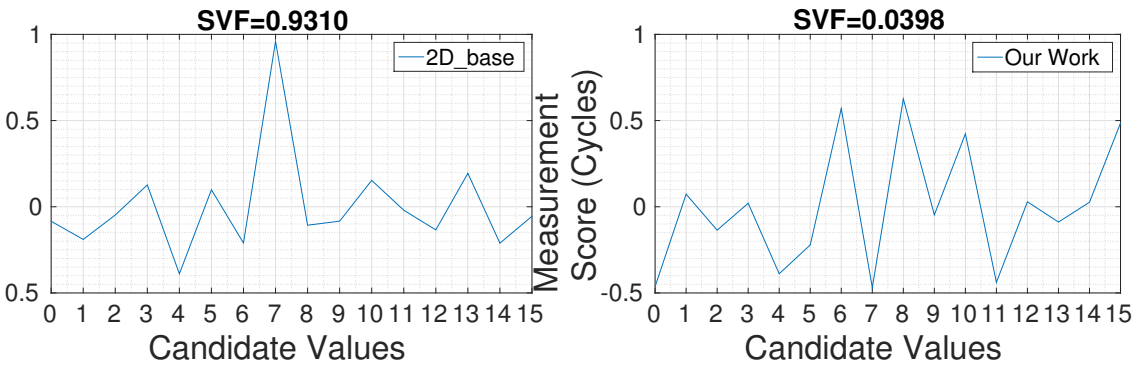


Figure 4.15: Measurement Scores for Deducing Key Byte 16 for Both Systems.

deduce that 7 is correct while in a system with our countermeasures applied, the attacker cannot gain any useful information. The SVF data for both systems are also shown on top of each plot. Clearly, there is over 95% reduction in SVF which shows that our system can defend against this kind of attacks successfully.

In the Prime+Probe attacks on LLC, we run a victim process that accesses cache set 81. We plot the reload time for cache set 40-100 for both systems in Figure 4.16. Similar to Prime+Probe attacks on L1, the attacker will see which cache set has the largest reload time and deduce that it is the cache set that the victim accesses. Clearly, on the 2D baseline system, the attacker can correctly deduce that cache set 81 was accessed by the victim. However, on the system with our countermeasures applied, the attackers measurements cannot give him too much useful information. If he has to make decision on which cache set was accessed by the victim, the best he can do is to pick the one with the largest reload time, which is cache set 92. Obviously, the attacker will make a mistake on this system. Therefore, we conclude that our system will prevent the attacker from launching Prime+Probe attacks on LLC.

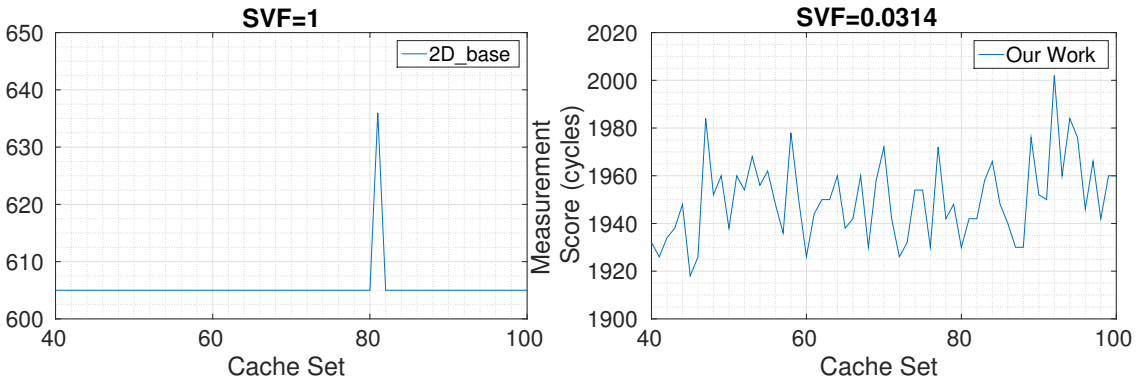


Figure 4.16: Reload Time for Cache Set 40-100 for Both Systems.

In time-driven attack, we plot the measurement scores (see [2,70]) for candidate values 0-31 when deducing key byte 8. The attacker guesses that the correct key is the one with the highest measurement score. Note that the true value of key byte 8 is 0x07 in this experiment. As shown in Figure 4.17, in neither systems, the attacker can recover the correct key directly. This is due to the fact that to successfully recover the key, usually time-driven attacks require a huge number of samples (for example, 2^{25} samples are used in [2]). However, when running on a Gem5 simulator, 2^{25} samples would take more than a month to simulate. Therefore, we only use 2^{22} samples in the simulation. With less samples, the attackers cannot directly recover the key. However, if he follows the steps in [2], he can quickly recover the key in the 2D_base system using a brute-force as candidate 0x07 has the second highest measurement score. However, in the system with our countermeasures applied, the brute-force attack would take a long time since candidate value 0x07 has very small measurement scores compared with others. Therefore, we conclude that our system will defend against time-driven attacks to some extent.

We plot the attackers measurements in the branch prediction attack (BPA)

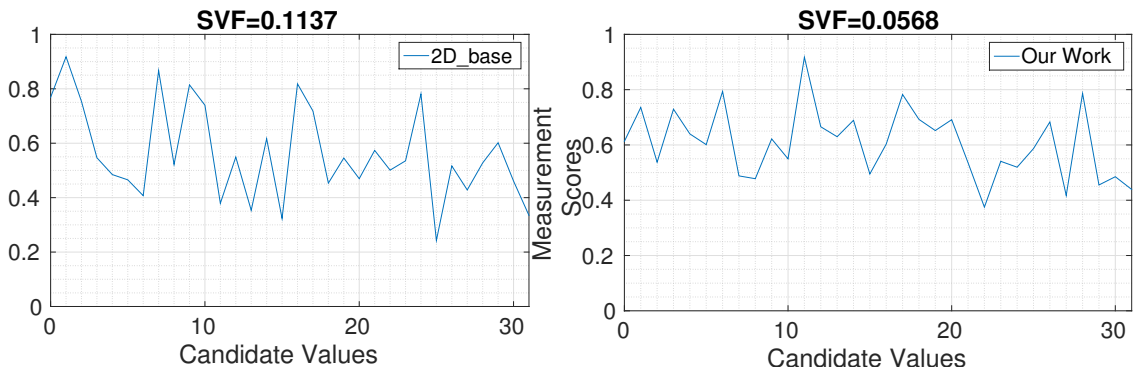


Figure 4.17: Attacker's Measurements in Time-driven Attacks.

on two systems in Figure 4.18. The attacker will deduce a 32-bit key based on these measurements. A long (short) execution time leads him to deduce that the corresponding bit is 1 (0). In both figures, measurements corresponding to key bit 1 are plotted in solid red dots while those corresponding to key bit 0 are plotted in open dots. In the right figure, the attacker can draw a clear threshold line between solid and open dots, meaning he can easily and correctly obtain the key. However, in the left figure, solid and open dots are mixed up and the attacker cannot learn the full key. These results show that our proposed work can defend against attacks on the branch prediction buffer effectively. We also note that in [60], the authors proposed to use BPA on attacking address space layout randomization. The underlying mechanism is exactly the same as the one used in the above attack. Therefore, our defense method could successfully prevent this attack from happening.

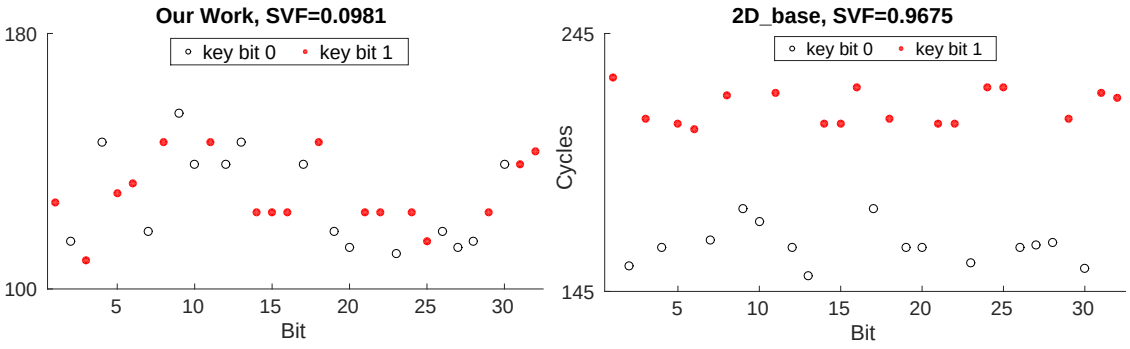


Figure 4.18: Attacker's Measurements in BPA.

4.6 Comparison with Related Work

We summarize the effectiveness of our proposed techniques and several other existing techniques against various attacks in Table 4.8. Note that no countermeasures can defend against all types of attacks. Different techniques need to work together to ensure a secure CPU. Our address permutation technique is the only one that can mitigate attacks on BTB. In addition, we investigate several vital features of a 3D CPU and show how they can be exploited to minimize timing side-channel leakage while still achieving some performance gain over a 2D baseline processor (see experimental results provided in Section 5.4).

In addition, we compare our proposed countermeasures with existing ones. Most partitioning-based approaches such as [72–74] have heavy performance overhead because they reserve part of the cache for the use of the secure process only. For example, the worst-case performance overhead was reported to be around 20% in [73]. Random eviction technique has been mentioned in several papers as a conceptual idea [55, 119]. However, neither implementation details nor performance or

Table 4.8: Comparison of Several Countermeasures

Methods	Access-driven attacks			Time-driven attacks
	L1	BTB	Others	
Claim 1&2	No	No	Yes	Yes
Addr Perm	Yes	Yes	Yes	No
[76]	Yes	No	Yes	Yes
[72, 73]	Yes	No	Yes	No

security evaluation is given. The random fill technique proposed in [76] has heavier performance overhead than the random eviction method we proposed. Countermeasures proposed in [77, 78] are not effective against timing-driven attacks. Moreover, going to 3D does not benefit the performance. The RPCache proposed in [77] has around 30% access latency overhead compared with a conventional set-associative cache. On the contrary, our proposed address permutation technique only incurs at most 1.7% performance overhead. The implementation logic in [77] is also much more complex than ours because of an additional level of indirection. It also consumes more area and power due to the usage of LNRegs which stores the cache-to-memory mapping information. The 3D-monitor based countermeasure proposed in [127] uses a monitor to control accesses to the cache owned by the victim process at the run-time. However, it has similar limitation with the cache-partition work [72].

4.7 Conclusion

In this chapter, we have proposed comprehensive methods to design a secure 3D CPU that has small timing side-channel information leakage. Performance and security evaluations show that compared with a 2D baseline processor, our proposed techniques are very effective against many widely-used timing side-channel attacks while still achieving remarkable performance gain.

Chapter 5: Side-Channel Attacks on Path-ORAMs

5.1 Introduction

It has been demonstrated that encryption of data alone is not sufficient to protect client's privacy when the client is accessing data in a remote storage. [81] shows that memory access patterns can leak very sensitive information even if the underlying data is encrypted. To mitigate this leakage, oblivious RAM (ORAM) has been proposed to conceal the actual access pattern from an adversary who is observing the accesses to the remote storage.

ORAM algorithm was first proposed by Golderich and Ostrovsky [82]. In the original form, it involved heavy shuffling, encryption and decryption operations, which had lots of performance overhead. Since its introduction, researchers have made significant progress in developing efficient and low-overhead ORAM protocols [83–87]. Most notably, in [88], Stefanov et al. proposed Path ORAM, a very efficient and simple ORAM protocol.

In [82, 88] and many subsequent efficient implementations of ORAMs, such as [128], the CPU is assumed to be secure and the ORAMs are proved to be zero-leakage from an adversary who is monitoring the memory traffic under this assumption. However one can surely imagine realistic situations where a piece of Malware

is running on the CPU along with secure code. In such a scenario, the ORAM's security needs to be evaluated. When the CPU is no longer secure, although ORAMs are proved to be secure against an attacker observing the memory traffic (referred to as an outside attacker), it is not secure any more against an adversary who can execute malicious programs on this CPU (referred to as an inside attacker), as we will show in this dissertation.

In this chapter, we evaluate the security level of ORAMs when the adversary can execute malware which observes the secure thread behavior through timing side channels. We identify common leakage points in many efficient ORAM protocols and propose several attack scenarios to demonstrate that the timing side-channel leakage exists. We hope that the analysis in this dissertation would motivate a new line of research to make ORAMs more secure to such attacks. We summarize our main contributions below:

- We identify three common leakage points in many recently proposed, efficient ORAM implementations.
- We propose different attack scenarios where an inside attacker and outside attacker collude to learn the underlying sensitive information and implement these attacks on some popular ORAM protocols.
- Experimental results on both FPGA and simulators are obtained and they demonstrate that the proposed attacks are feasible and can leak sensitive information.
- Possible mitigation methods are provided and discussed.

1: Read private data D, Allocate an array T, $counter \leftarrow 0$

2: **for** $i = 1; i < |D|; i++$ **do**

3: **if** D[i]=1 **then**

4: $T[counter++] \leftarrow 1$

5: **else**

6: Do something else

7: **end if**

8: **end for**

Figure 5.1: Algorithm1: Trusted Program

The rest of the chapter is organized as follows: details on how different attacks can be performed on the ORAMs are given in Section 5.2. Section 5.3 outlines several possible countermeasures to mitigate the proposed attacks on ORAM. Experimental results are listed and relevant discussions are given in Section 5.4. Finally, Section 5.5 concludes the paper.

5.2 Timing Side-channel Attacks on ORAMs

5.2.1 Previous Work

One timing side-channel attack on ORAM was demonstrated in [129]. The trusted client's program is shown in Figure 5.1. It takes a client's private input D and based on each bit of D, it either makes an access to the memory or not. An attacker observing the processor's output pins can learn D effectively. At the time

the trusted program is processing bit i of D, if the attacker observes that there is an ORAM access, he knows that $D[i] = 1$. Otherwise, he knows that $D[i] = 0$. The root cause of this attack is that *when* ORAM is accessed leaks privacy. Note that possible countermeasures have been dicussed in [129]. The idea is to use a queue to store all incoming ORAM requests and visit remote storage in a fixed interval (or issue a dummy request if the queue is empty). This idea has been adopted by other ORAM researchers [128, 130, 131] and we assume that it has been implemented in all ORAMs.

5.2.2 Attack Model

We make the following assumptions about the attack model. **Multi-core**. We assume that the processor is multi-core. When each core has a miss in the last-level of cache, it places a request to the ORAM. The processor is running one trusted program on one core.

Outside attacker. An attacker can monitor the processor's output pins. We refer to this attacker as an outside attacker. More specifically, he can observe the following: 1) when the program is loaded onto the processor and terminates; 2) the address sent to the external storage and the associated data; 3) when each access to external storage is made. This outside attacker is the adversary model often used in the literature when assessing the security of the ORAM [84, 88]. ORAM is provably secure to an outside attacker working alone.

No privilege. The inside attacker has no access to privileged instructions and

cannot bypass the restrictions imposed by the operating system, such as reading another program's memory. This assumption is also in accordance with [129].

Above are common assumptions used in the ORAM protocols. In addition, we make two more realistic assumptions:

Inside attacker. Another attacker can execute untrusted programs on the CPU. He can run his program either before, after, or simultaneously with the trusted program. We refer to this attacker as an inside attacker. The inside attacker may not necessarily be the same person as the outside attacker but they will collude to figure out the underlying private information. Even if no trusted program is running, these attackers may still collude to learn the ORAM implementation characteristics, which could be used later to attack a trusted thread.

Timing measurements. We assume that the inside attacker can measure, in his program, the time between when a data request is made and when the data is ready. We believe that this is a valid assumption since most modern processors provide instructions to read the internal timestep counter (such as RDTSC in X86 processors). The inside attacker can utilize these instructions to obtain the above timing information.

Using the above attack model, we note that the attack [129] shown in Section 5.2.1 and ours have salient differences. We assume that the CPU is not secure and our attacks utilize the timing side-channel information measured by both inside and outside attackers while [129] only relies on the outside attacker to measure timing side-channel information. As a result, our attacks are more powerful and harder to mitigate. In the next several sections, we will show how the ORAM protocol can

be compromised with the help from an inside attacker.

5.2.3 Attacks on Queuing Delay

To obfuscate *when* each ORAM access is made, a queue is used to store all the incoming requests, ORAM is accessed a fixed periodic rate. This seems to be working in the single-core environment. However, in a multi-core scenario, this queue will become the source of the leakage, as demonstrated next.

Consider the scenario that the trusted program visits a memory location that only it has access to and retrieves some private data D. The trusted program has a memory access pattern that is dependent on the private data D (e.g., Figure 5.1). The attacker's goal is to learn D. However, he cannot learn it directly as the operating system prohibits him from reading another program's memory.

The outside attacker alone cannot learn D by observing from the processor's output pins only. Because if the ORAM is not accessed, a dummy request will be sent after some interval to satisfy the requirement that ORAM is accessed at a fixed rate. Therefore, from the perspective of the outside attacker, access to external storage is always happening, regardless of the secret data. This means that an outside attacker alone cannot learn anything. However, as mentioned above, in the multi-core setting, the queue used to store ORAM access requests may become the source of leakage. What the attacker does is to run the attack program shown in Figure 5.2. The inside attacker keeps sending requests to the ORAM (line 4 in Algorithm 5.2). He measures the time it takes to fulfill his request (line 3 in

1: Allocate a piece of memory A, $counter \leftarrow 0$

2: **while** NOT TERMINATED **do**

3: $t1$ =RDTSC(), $A[counter++] = 1$, t =RDTSC()-$t1$

4: **end while**

Figure 5.2: Attack Program 1

Figure 5.2). If there is nothing in the queue, then his request will be serviced quickly. Otherwise, if the trusted program also accesses the ORAM, then there will already be a request in front of the attacker's request, causing the service of his request to be delayed. Thus, by observing the request completion time, the inside attacker can learn D.

The amount of information leakage depends on the synchronization between the attacker's measurements and each bit of D. To leak full bits of D, the inside attacker's program needs to be executed before the start of the trusted program. Moreover, the attacker needs to know exactly when the trusted program is processing each bit of D. This can be learned from an offline profiling of the trusted program or from collusion with the outside attacker who can observe certain processor pins. When such information is not available to the inside attacker, he can still leak partial information about D. Such partial information may greatly reduce the exploration space when brute-forcing D and we consider this attack meaningful.

5.2.4 Attacks on Encryption and Decryption Time

As introduced in Section 2.3.2, a majority of time spent in the ORAM controller is to decrypt the data blocks retrieved from the memory. To speed up the process, one implementation (such as Ascend architecture [131]) will be attempted to decrypt and store in the stash only the real blocks (and ignore those dummy blocks). This will still be secure against an outside attacker alone. However it will be problematic in the presence of an inside attacker. Consider the following scenario. The trusted program waits for user's private input D and writes data to either an old location that it has visited or a new location depending on D. The attacker tries to guess the sequence of these memory visits and from there, he can learn D. For example, the trusted program has visited block 1-5. Upon receiving the next input D, it either visits an old memory location (block 1-5) or a new one (block 6). Knowing which block the trusted program visits reveals the private input D. However, the outside attacker alone cannot achieve this. The memory will still bring the whole paths from the requested leaf to the root to the stash, thus making the memory trace exactly the same as original Path ORAM protocol. Then those dummy blocks will be ignored during the decryption process in the ORAM controller. This will cause some timing difference depending on how many real blocks there are in the retrieved path. However, since the ORAM controller has an outgoing queue and will access the remote storage in a uniform rate, this timing difference will be masked by the effect of the outgoing queue. From the perspective of the outside attacker, it is the same as if both real and dummy blocks were decrypted. Therefore, from the

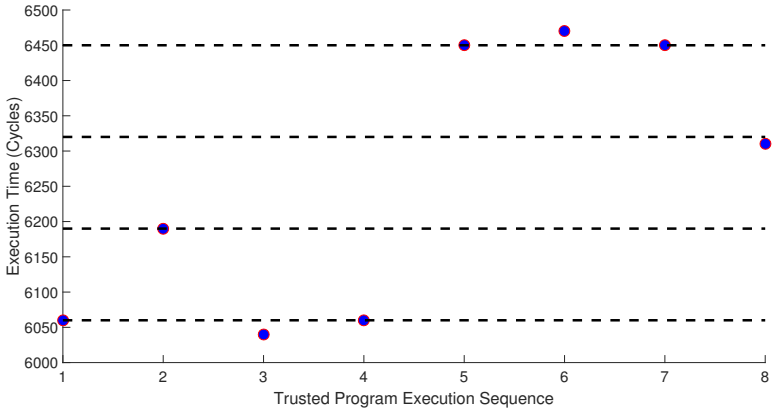


Figure 5.3: Attacker's measurements given by the FPGA.

perspective of an outside attacker alone, the above implementation is still secure.

On the other hand, the inside attacker can deduce how many real blocks are in each path by observing the execution time of the trusted program. We use one motivational example to demonstrate this. We simulate the execution of one trusted program on a 5-level ORAM on a FPGA board. Intially, the ORAM is empty. The trusted program is then executed eight times and writes to block 1, 2, 3, 4, 2, 4, 3, 4 during each execution. Note that these blocks may reside in any bucket (1-31 shown in Figure 5.4). The inside attacker measures the program execution time of each run and the results are shown in Figure 5.3. Section 5.4 gives details on simulation framework.

Note that the execution time of run1 and run 2 varies by roughly 130 cycles. Since the ORAM is originally empty, the attacker knows that 0 blocks are decrypted during run 1 and 1 block during run 2. Using the same logic, the attacker can see that 0, 1, 0, 0, 3, 3, 3, 2 real blocks are decrypted in each run. Next, we demonstrate that if the outside attacker and the inside attacker collude, they can derive the whole

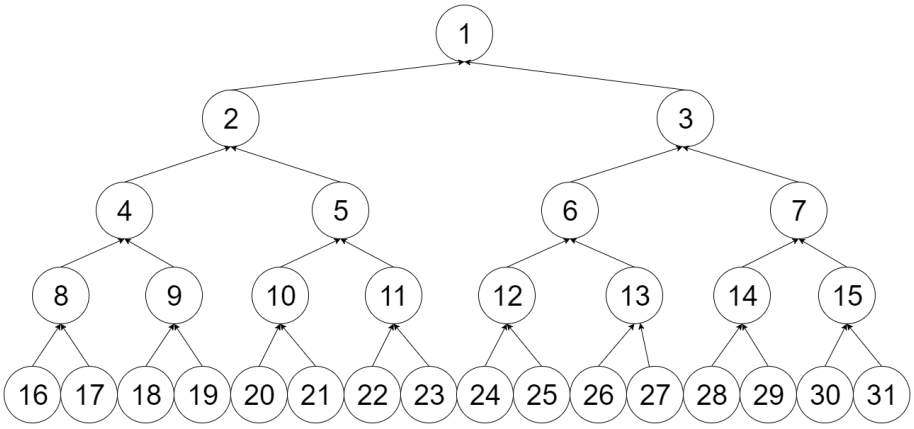


Figure 5.4: One ORAM structure. Each node is a bucket containing multiple blocks.

execution sequence or greatly reduce the search space for the execution sequence. They can even guess where each block is with high confidence. For example, suppose that the outside attacker observes that leaf bucket 22, 27, 16, 20 (see Figure 5.4) are accessed during the first four executions. In run 2 (where leaf bucket 27 was accessed), the inside attacker observes that one real block is decrypted. The outside attacker observes that path 22 is accessed in the first timestep and path 27 is accessed in the second one. Since the only intersection between path 22 and 27 is bucket 1, and one real block resides in path 27, it can be concluded that the real block added in the first timestep was in bucket 1 at the end of first execution of the trusted program. Then in the third and fourth run, the inside attacker observes that no real block is decrypted. Therefore, he concludes that run3 and run4 each writes to a new location. Otherwise, there should be at least one real block in the path to be decrypted. Deducing the rest of the sequence is similar. We have developed an attacker program that takes as input the number of real blocks decrypted during each visit (observed by the inside attacker) and the associated leaf bucket that

has been visited (observed by the outside attacker), it will output all the possible memory access sequence and possible locations of each block after each visit. The idea is to enumerate the possible location of each block after each execution, and remove those locations that conflict with the observation of either the inside attacker or the outside attacker. Our attacker program on the above example outputs 428 possible memory access sequences. Compared with 8! possible sequences if only the observation of the outside attacker is available, we are able to reduce the search space by $1 - 428/8! = 98.93\%$.

5.2.5 Attacks on Using Stash as Cache

To make ORAM more efficient, [132] proposes to use stash as a cache for local read/write operations. When the client wants to read data that is in the stash, the client will directly read from the stash rather than accessing the server. Though this technique can increase the efficiency of ORAM, it has serious security issues in the context of timing side-channel attacks originating from inside, as illustrated next.

Consider the following attack scenario. The trusted program runs on a private key which is unknown to the attacker. The trusted program has a memory access pattern that depends on the private key (for example, it visits the same block it visited last time if the key bit is 0, or it visits a new block if the key bit is 1). The adversary is trying to deduce the private key. The use of ORAM is proved to be secure against an outside attacker alone. However, the inside attacker can run the trusted program and measures how long it takes to finish. If the total execution time

is smaller, then it is highly likely that the trusted program has a repeated sequence of memory visits that result in cache hits.

To explain this attack in more detail, let us consider two different ORAM protocols that have such caching behavior. In Phantom [128], tree-top caching is proposed, which caches the content of blocks in the first few layers of the path ORAMS (starting from the root) in the stash. For example, one implementation chooses to store all blocks that should have been stored in bucket 1-7 in the stash. The number of layers to cache is a design parameter that affects performance. The more layers to cache, the higher probability of cache hits there is. We use the ORAM structure shown in Figure 5.4 to illustrate the leakage. Note that the actual ORAM may have a much higher level than the structure shown. For illustration purpose, let us assume that in one implementation, three layers of blocks are stored in the stash. This means blocks in bucket 1-7 will be cached in the stash. To help our illustration, we define $A_i(j)$ as the ancestor of bucket j in the ith level. For example, in Figure 5.4, $A_1(3) = 1$ and $A_3(20) = 5$. Consider what will happen if a block (say block1) has been repeatedly visited. If block1 is stored in a bucket i where $A_4(i) = 8$ (e.g., bucket 16) in the first visit, during next visit of block1, the ORAM protocol dictates that it be remapped to leaf bucket 16-31 with equal probability. This means, with probability 7/8, it will be remapped to a bucket i' where $A_4(i') \neq 8$. Then according to ORAM protocol, block1 will be relocated to either bucket 4 or 2 or 1 and will be cached in the stash since it is now in the first three levels of the ORAM. As a result, with a probability of 7/8, the block will be cached and the same visit to this block next will be a cache hit and hence very short access time. On the other

hand, if a different block (say block2) is visited after visiting block1, the access time of block2 will be independent of that of block1. In summary, if the program has an access pattern that visits a different block each time, then on average, the access time of each block visit is roughly the same. However, if the program keeps visiting the same block, then there is a high probability that the next block visit is a cache hit, resulting in a shorter execution time. Therefore, if the attacker sees the average execution time of two memory sequences, he may conclude that the sequence that has a shorter execution time contains many consecutive visits to the same blocks.

In Fork-path ORAM [133], all blocks along the path visited during an ORAM access are cached in the stash. This scheme is very vulnerable against an inside attacker. Using the same logic, we can argue that if the program visits two different blocks, the access time will be independent (the acccess time of the second block visit may or may not result in a cache hit). However, if the program visits two same blocks, then the second block visit is guaranteed to result in a cache hit, since the path visited during the second visit is guaranteed to be overlapped with the path in the first visit, according to the invariant of the path ORAM. Therefore, same as above, an attacker can tell if a sequence contains consecutive visits to the same block by measuring the program execution time.

5.3 Mitigation Techniques

Generally, one might be tempted to prohibit the use of rdtsc instruction since it is used in the attack to measure time. However the trusted program running on

the processor may need such instruction for performance profiling purpose or other purposes. Even without this instruction, the attacker can connect a timing source to the processor and use it to measure the timing information (like what we did in the FPGA board simulation). Therefore, we conclude that prohibiting the use of rdtsc instructions is neither possible nor enough.

To defend against the attack on queuing delay proposed in Section 5.2.3, we can make the external storage multi-port, meaning that it can be accessed through different ports simultaneously. One queue for each core is used. The requests in different queues are sent to different ports of the external storage. In this way, the requests generated by one core will not interfere those generated by another core and thus eliminating the side-channels. A similar idea is to use a different ORAM controller for each core. However, these may incur area overhead as extra queues and controllers are needed.

To defend against the attacks on the decryption and encryption time shown in Section 5.2.4, the protocol should be modified. More specifically, we identify the root cause of the leakage as decrypting ONLY the real blocks. Therefore, to defend against this attack, all blocks including dummy blocks should be decrypted. Security vs. performance trade-off will be presented in Section 5.4. For using stash as cache, the trade-off between security and performance is given in Section 5.4. The system designer should choose a design according to security/performance need.

5.4 Experiments and Results

5.4.1 Experimental Setup

ORAM protocols implemented. To demonstrate that our proposed attacks are effective, we implement several newly proposed Path-ORAM protocols. More specifically, we implement the ORAM used in the Ascend architecture as proposed in [131]. Tree-top caching [128] as well as Fork-path ORAM [133] are also implemented. These ORAM protocols are provably secure against an outside attacker and they optimize ORAM bandwidth and performance in various ways. However, we demonstrate that they are vulnerable to an inside attacker.

Attacks performed. We will implement the attacks introduced in Section 5.2.3, 5.2.4 and 5.2.5 on the ORAM protocols mentioned above. More specifically, attacks on queuing delay and attacks on decryption time are applied to the Ascend ORAM while attacks on caching behavior are applied to Phantom and Fork-path ORAM.

Simulator. To demonstrate that our proposed leakage does exist, we write an ORAM simulator for each of the above ORAM protocol. We run some experiments on the simulators to demonstrate the effectiveness of the proposed attacks. All the parameters used in our simulators are listed in Table 5.1. We assume that the external storage is a conventional DRAM with 200-cycle latency. The encryption and decryption is done using random AES encryption and is done at a speed of 100 cycles/block [134]. Since stash usually only holds a small portion of overflow data,

Table 5.1: Parameters Used in the Simulator

Number of Bucket Per Node, Z	4
Number of Levels $L+1$	10
Block Size	128 Bytes
External Storage Latency	200 Cycles
Encryption/Decryption Time	100 Cycles
Stash Latency	30 Cycles

we assume that it is SRAM and has a latency of 30 cycles. We also enforce the ORAM to be accessed at a fixed interval as proposed in [134], i.e. 100 cycles.

FPGA-based results. To show that our attacks are real, we also implement the ORAM protocols used in the Ascend architecture [131] on FPGA. We choose the Altera Stratix V FPGA, a high-end FPGA with on-board 2GB DRAM (shown in Figure 5.5). We modify the Altera NIOS-II architecture to add an ORAM controller between the CPU and the DRAM. The attacker program is developed in C and run on the NIOS-II software-IP. A high-resolution counter is provided as the attacker's source to measure time. We then obtain the results for attacks on decryption time.

5.4.2 Attacks on Queuing Delay

This subsection evaluates the effectiveness of the attack on queuing delay. We randomly generate fifty values of the secret data D. For each D, we start the inside attacker's program on the simulator first. Then we run the trusted program. We assume that processing each bit of D takes 3000 cycles and this is known to the attacker. The attacker's measurements corresponding to two values of D are given

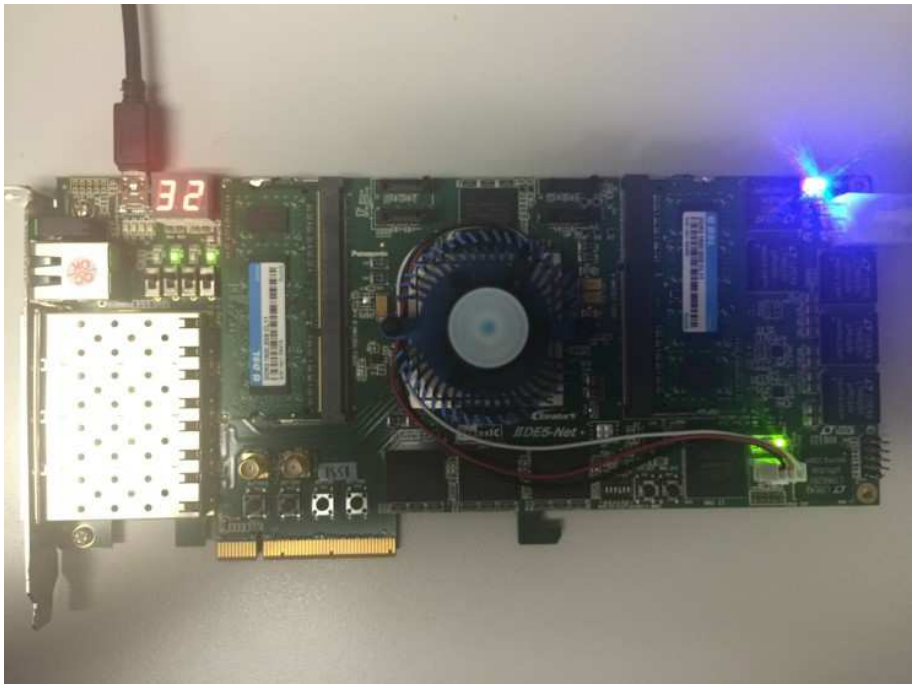


Figure 5.5: Stratix V FPGA board used in the experiment.

by the simulator and shown in Figure 5.6. 0 in the x-axis corresponds to the start of the trusted program. Each bin in the figure corresponds to each bit of D. If the attacker observes that the fulfillment of his data request takes longer in one bin, he knows that the corresponding bit of D is 1. Figure 5.6 shows clearly that such attack is able to leak D successfully. In fact, in all the fifty D generated, the inside attacker is able to learn D successfully.

5.4.3 Attacks on encryption and decryption time.

We use the FPGA board to execute the trusted program introduced in Section 5.2.4. Each time, we first initialize the ORAM to empty and run the trusted program 8 times, visiting either old or new locations randomly. The board outputs for each execution of the trusted program, which leaf node is visited and how long it

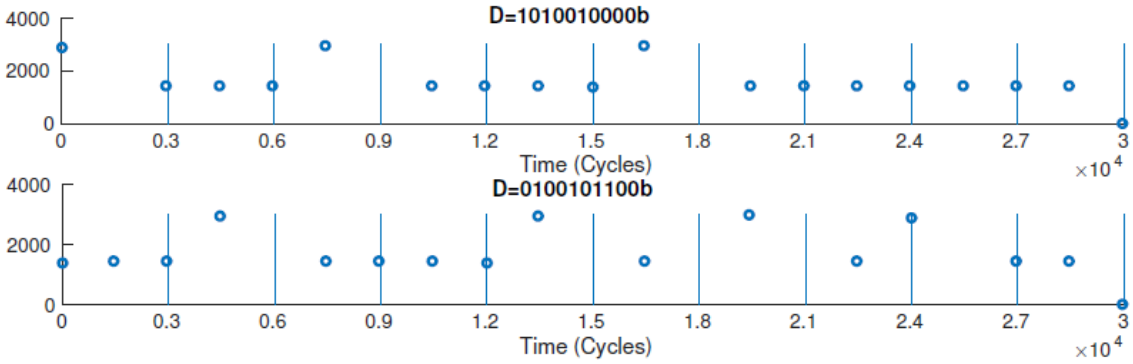


Figure 5.6: Attacker's measurements for two different values of D. If fulfillment of attacker's data takes longer in one bin, he knows the corresponding bit in D is 1.

takes to finish. This represents the measurements from both the inside and outside attacker. The attacker program introduced in Section 5.2.4 is then run to deduce the memory access sequence as well as the possible locations of each block after each memory visit. The above steps are repeated 100 times and the results are averaged. To gain a sense of how powerful our attack program is, we record the run time of our attacker program, which represents how quickly can our program deduce the subset of possible memory access patterns. The results are averaged over 100 runs and shown in Table 5.2. For comparison purposes, we also list the number of possible access sequences and number of possible locations if given only the observation from the outside attacker. We can see from Table 5.2 that our attacker program can reduce the search space by orders of magnitude.

5.4.4 Attacks on Caching Behavior

We run the attack described in Section 5.2.5 on both Phantom and Fork-path ORAMs on the simulator. In Phantom, both caching only the root and caching the

Table 5.2: Evaluation of our attacker program.

Attacker's Information	Inside+Outside Attacker	Only Outside Attacker
Run time	10m14s	-
#Possible Access Sequences	386	40320
#Possible Locations	10475520	$1.73 * 10^{14}$

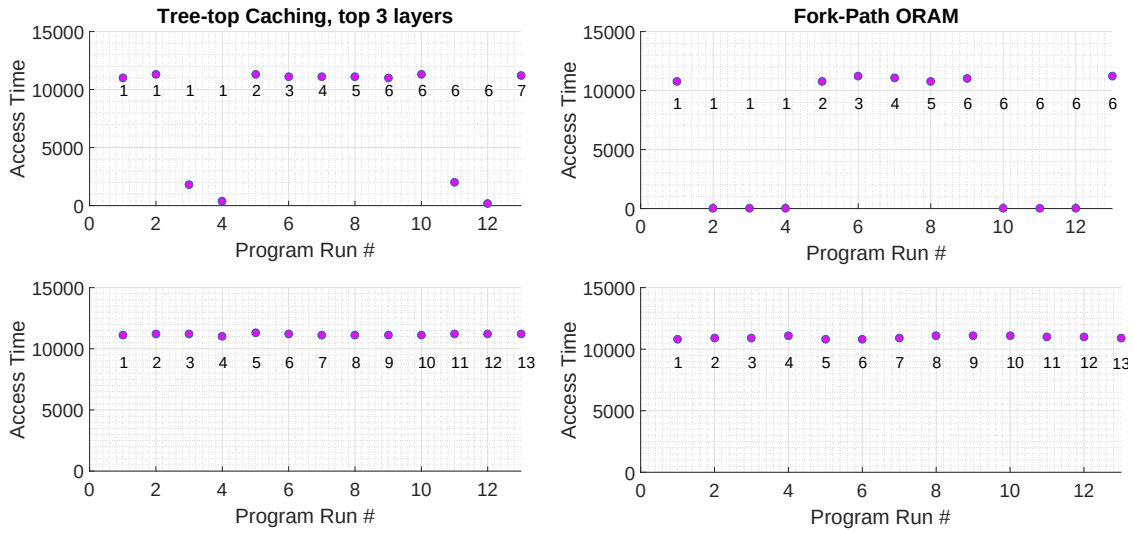


Figure 5.7: Attacks on using stash as cache. Left 2 figures show tree-top caching and right 2 figures show fork-path ORAM. Block access sequences are denoted.

first three layers are explored. We simulate two different memory access sequences on the simulator 100 times. We plot the averaged execution time of each memory access given by the simulator in Figure 5.7. From the figure, we can tell that if we keep visiting the same block, the subsequent access time of the same block in both protocols will be smaller than that of visiting a different block. In Fork-path ORAM, since visiting the same block is guaranteed to be a cache hit, hence near-zero access time, the leakage is the greatest.

Table 5.3: Evaluation of decrypting all blocks.

What to Decrypt	Run-time of 50 Memory Visits (cycles)	Correlation
All	648200	0.1651
Only Real	607200	0.9867

5.4.5 Evaluation of Mitigation Techniques

We evaluate mitigation techniques in Section 5.3. We first compare the performance and security of decrypting all vs. only real blocks in Path-ORAMs. We initialize the ORAM to empty, visit 50 random blocks, and record the time to finish these 50 memory visits. We repeat this step 100 times and the averaged time is shown in Table 5.3. For security evaluation, we calculate the correlation coefficient between the number of real blocks decrypted and the execution time of the trusted program. The averaged results are also shown in Table 5.3. We can see that with an average of 6.33% performance downgrade, we can achieve over 6X correlation reduction. Therefore, decrypting all blocks should always be used.

We then compare the performance and the security of treetop caching. Specifically, we compare the effect of caching only the first 1/2/3 layers and no caching. For each case, we randomly generate 50 memory visits, each either visits the same block as previous request or visits a new one. We record the time it takes to finish these 50 memory visits. The above steps are repeated 100 times and the averaged time is shown in Table 5.4. For security evaluation, we calculate the correlation coefficient between the execution time of the trusted program and a binary indicator $\mathbf{I_{same}}$, where $\mathbf{I_{same}} = 1$ indicates that the same block is visited as the previous visit

Table 5.4: Evaluation of different caching behavior.

# Layers to Cache	Run-time of 50 Memory Visits (cycles)	Correlation
0	609300	0.1302
1	412600	0.9467
2	346800	0.9732
3	307500	0.9893

and $\mathbf{I_{same}} = 0$ indicates a different block is visited. The averaged results are also shown in Table 5.4. The performance of caching top 3 layer almost doubles compared with no-caching. However, the security is 7X worse. The reason is as follows. As we increase the number of layers to be cached, the probability of a cache hit (stash hit) increases and hence the performance goes up. However, higher cache hit means visiting the same block twice takes shorter because the second visit is more likely to be a cache hit. As a result, the difference in latencies between visiting the same block twice and visiting two different blocks is greater. Therefore, the security goes down. We conclude that the use of caching should be decided according to the performance and security need of the design.

5.5 Conclusion

In this chapter, we have proposed several timing side-channel attacks on ORAMs. We demonstrate that ORAM is not secure against an inside attacker who executes untrusted programs and measures the timing information. We demonstrate that our proposed attacks can leak significant amount of information using exper-

iments on both simulator and FPGA. We hope that the analysis in this chapter would motivate a new line of research to make ORAMs more secure to the proposed attacks.

Chapter 6: Conclusion and Future Research Directions

In Chapter 1, we highlighted some of the key issues and challenges in hardware security. We demonstrate that techniques that enhance cyber security is urgently needed. We then introduce the general life cycle of a hardware system, including design phase, fabrication phase, test phase and post-deployment phase. Various hardware attacks that take place at each phase are also introduced. Existing design and verification techniques to mitigate these attacks are surveyed and their disadvantages are discussed. The main goal of our work is to overcome the challenges in solving hardware security problems. We also investigate the impact of emerging technologies and new hardware primitives on hardware security, including new countermeasures that are enabled by them and new security problems they bring. Chapter 2 introduces existing research problems and state-of-art results on hardware Trojan detection, timing side-channel attacks on cache, and Oblivious RAM.

In Chapter 3, we proposed a novel reverse-engineering based hardware Trojan detection approach without generating a gate or transistor netlist. Our approach adapts one-class ν-SVM to the HT detection problem. The experimental results on several publicly available benchmarks show that our method can achieve very high Trojan detection accuracy. We also investigated the impact of some modeling

and algorithm parameters on the accuracy rate. Our method is efficient in storage space, does not require the existence of golden chip and is robust to variations in fabrication and reverse-engineering process. To extend this work, we proposed a novel design-time strategy to aid test-time Trojan detection. Experimental results on real benchmarks showed that using our design-time strategy, we can detect on average 16.87% more Trojans with only 7.87% area overhead and 17.72% leakage power overhead. Our method is fully automated, can easily fit into the current design flow of IC and thus is very promising.

In Chapter 4, we proposed comprehensive methods to design a secure 3D CPU that has small timing side-channel information leakage. We explored the implications of 3D integration technique on cache timing side-channel signatures. We investigated how 3D integration can be used to mitigate cache timing side-channel attacks. Performance and security evaluations showed that compared with a 2D baseline processor, our proposed techniques are very effective against many widely-used timing side-channel attacks while still achieving remarkable performance gain.

In Chapter 5, we proposed several timing side-channel attacks on ORAMs. We demonstrated that ORAM is not secure against an inside attacker who executes untrusted programs and measures the timing information. We showed that our proposed attacks can leak significant amount of information using experiments on both simulator and FPGA. We hope that the analysis in this dissertation would motivate a new line of research to make ORAMs more secure to the proposed attacks.

6.1 Future Work

In this dissertation, we have proposed some innovative and unconventional approaches to enhance hardware security. There are still opportunities for further improvements and open issues such as the implication of emerging technologies on other types of hardware attacks.

Implications of 3D integration on other hardware attacks. In this dissertation, we explored the implications of 3D integration on timing side-channel attacks on cache. As pointed out by Xie et. al [135], 3D integration will bring many opportunities and challenges in making hardware more secure. In our future work, we would like to investigate the implications of 3D integration on other hardware attacks, including:

- Hardware Trojan detection. Researchers have proposed several design-for-security techniques to defend against hardware Trojans. These techniques usually insert some circuitry into the original circuit to function as the defense mechanism or aid test-time Trojan detection. However, in conventional IC fabrication model, these additional circuitry are also fabricated at the untrusted foundry, leaving their security in doubt. Attackers could sabotage them, leaving designer's effort in vein. 3D integration technique, on the other hand, enable designer to integrate these design-for-security circuitry as a separate die and manufacture them in a trusted foundry. As such, the security of these additional circuitry is guaranteed. In addition, 3D integration techniques also allow designer to integrate run-time monitor on top of the original circuit.

Since vertical interconnection (between dies) is much shorter than horizontal interconnection (within a die), the performance overhead can be reduced.

- Reverse-engineering attacks. Because 3D integration process is more complicated than conventional fabrication process, the complexity for attackers to reverse-engineer an IC also increases. This makes 3D integration a natural defense against reverse-engineering attacks. Moreover, 3D integration enables designers to include active-shielding of security-sensitive areas with additional layers of metal. The active shielding techniques make the devices tamper-resistant.

- Fault-injection attacks. The stacking structure of 3D integration offers a natural defense against fault injection attacks. The first phase of fault injection attacks is to inject faults into the device using light illumination, focused ion beam, etc. By placing the vulnerable tier (e.g.,,, memory) in the lower stack, the stack above it will act as a shield, which may protect the vulnerable tier from being influenced by fault injection techniques. Furthermore, the stacking structure enables system designers to stack a monitor tier on top of the original system. The monitor tier may watch the system for any intended faults.

ORAM security. In this dissertation, we demonstrated that timing side-channel attacks can break ORAM, a new security-oriented hardware primitive. Though we have introduced some general guidelines in designing a secure ORAM in this dissertation, it merely scratched the surface of the ORAM design-space exploration. How to balance the trade-off between performance and security is the central topic

to explore. More experimental results are needed to better design a secure and efficient ORAM protocol.

Bibliography

[1] Yuval Yarom and Katrina E Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

[2] Daniel J Bernstein. Cache-timing attacks on aes, 2005.

[3] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[4] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912. USENIX Association, 2015.

[5] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320. ACM, 2007.

[6] Alex Baumgarten, Michael Steffen, Matthew Clausman, and Joseph Zambreno. A case study in hardware trojan design and implementation. *International Journal of Information Security*, 10(1):1–14, 2011.

[7] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *Design & Test of Computers, IEEE*, 27(1):10–25, 2010.

[8] Ramesh Karri, Jeyavijayan Rajendran, and Kurt Rosenfeld. Trojan taxonomy. In Mohammad Tehranipoor and Cliff Wang, editors, *Introduction to Hardware Security and Trust*, pages 325–338. Springer New York, 2012.

[9] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.

[10] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference*, pages 83–89. ACM, 2012.

[11] Jeyavijayan JV Rajendran and Siddharth Garg. Logic encryption. In *Hardware Protection through Obfuscation*, pages 71–88. Springer, 2017.

[12] Jiliang Zhang. A practical logic obfuscation technique for hardware security. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(3):1193–1197, 2016.

[13] Yu-Wei Lee and Nur A Touba. Improving logic obfuscation via logic cone analysis. In *Test Symposium (LATS), 2015 16th Latin-American*, pages 1–6. IEEE, 2015.

[14] Farinaz Koushanfar. Provably secure active ic metering techniques for piracy avoidance and digital rights management. *IEEE Transactions on Information Forensics and Security*, 7(1):51–63, 2012.

[15] Farinaz Koushanfar and Yousra Alkabani. Provably secure obfuscation of diverse watermarks for sequential circuits. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*, pages 42–47. IEEE, 2010.

[16] Jeyavijayan Rajendran, Michael Sam, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of integrated circuit camouflaging. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 709–720. ACM, 2013.

[17] Meenatchi Jagasivamani, Peter Gadfort, Michel Sika, Michael Bajura, and Michael Fritze. Split-fabrication obfuscation: Metrics and techniques. In *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*, pages 7–12. IEEE, 2014.

[18] Kaushik Vaidyanathan, Renzhi Liu, Ekin Sumbul, Qiuling Zhu, Franz Franchetti, and Larry Pileggi. Efficient and secure intellectual property (ip) design with split fabrication. In *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*, pages 13–18. IEEE, 2014.

[19] Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. Is split manufacturing secure? In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1259–1264. EDA Consortium, 2013.

[20] Frank Imeson, Ariq Emtenan, Siddharth Garg, and Mahesh V Tripunitara. Securing computer hardware using 3d integrated circuit (ic) technology and split manufacturing for obfuscation. In *USENIX Security*, volume 13, 2013.

[21] Gustavo K Contreras, Md Tauhidur Rahman, and Mohammad Tehranipoor. Secure split-test for preventing ic piracy by untrusted foundry and assembly. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on*, pages 196–203. IEEE, 2013.

[22] Peng Gu, Shuangchen Li, Dylan Stow, Russell Barnes, Liu Liu, Yuan Xie, and Eren Kursun. Leveraging 3d technologies for hardware security: Opportunities and challenges. In *Proceedings of the 26th edition on Great Lakes Symposium on VLSI*, pages 347–352. ACM, 2016.

[23] Yang Xie, Chongxi Bao, and Ankur Srivastava. Security-aware design flow for 2.5 d ic technology. In *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*, pages 31–38. ACM, 2015.

[24] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. A formal study of power variability issues and side-channel attacks for nanoscale devices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 109–128. Springer, 2011.

[25] Washington Cilio, Michael Linder, Chris Porter, Jia Di, Dale R Thompson, and Scott C Smith. Mitigating power-and timing-based side-channel attacks using dual-spacer dual-rail delay-insensitive asynchronous logic. *Microelectronics Journal*, 44(3):258–269, 2013.

[26] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 142–159. Springer, 2013.

[27] Kris Tiri. Side-channel attack pitfalls. In *Proceedings of the 44th annual Design Automation Conference*, pages 15–20. ACM, 2007.

[28] Ujjwal Guin, Daniel DiMase, and Mohammad Tehranipoor. Counterfeit integrated circuits: detection, avoidance, and the challenges ahead. *Journal of Electronic Testing*, 30(1):9–23, 2014.

[29] Ujjwal Guin, Ke Huang, Daniel DiMase, John M Carulli, Mohammad Tehranipoor, and Yiorgos Makris. Counterfeit integrated circuits: a rising threat in the global semiconductor supply chain. *Proceedings of the IEEE*, 102(8):1207–1228, 2014.

[30] Mark Marshall. Best detection methods for counterfeit components. *SMTA publication (presentation)*, 2011.

[31] Masoud Rostami, Farinaz Koushanfar, Jeyavijayan Rajendran, and Ramesh Karri. Hardware security: Threat models and metrics. In *Proceedings of the International Conference on Computer-Aided Design*, pages 819–823. IEEE Press, 2013.

[32] Farinaz Koushanfar, Saverio Fazzari, Carl McCants, William Bryson, Matthew Sale, Peilin Song, and Miodrag Potkonjak. Can eda combat the rise of electronic counterfeiting? In *Proceedings of the 49th Annual Design Automation Conference*, pages 133–138. ACM, 2012.

[33] Ujjwal Guin and Mohammad Tehranipoor. On selection of counterfeit ic detection methods. In *IEEE north atlantic test workshop (NATW)*, 2013.

[34] Ke Huang, John M Carulli, and Yiorgos Makris. Parametric counterfeit ic detection via support vector machines. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2012 IEEE International Symposium on*, pages 7–12. IEEE, 2012.

[35] Seetharam Narasimhan, Dongdong Du, Rajat Subhra Chakraborty, Somnath Paul, Francis Wolff, Christos Papachristou, Kaushik Roy, and Swarup Bhunia. Multiple-parameter side-channel analysis: A non-invasive hardware trojan detection approach. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*, pages 13–18. IEEE, 2010.

[36] Dongdong Du, Seetharam Narasimhan, Rajat Subhra Chakraborty, and Swarup Bhunia. Self-referencing: A scalable side-channel approach for hardware trojan detection. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 173–187. Springer, 2010.

[37] Michael Bilzor. 3d execution monitor (3d-em): Using 3d circuits to detect hardware malicious inclusions in general purpose processors. In *Proceedings of the 6th International Conference on Information Warfare and Security*, page 288, 2011.

[38] Xiaotong Cui, Kun Ma, Liang Shi, and Kaijie Wu. High-level synthesis for run-time hardware trojan detection and recovery. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.

[39] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Hardware trojan: Threats and emerging solutions. In *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*, pages 166–171. IEEE, 2009.

[40] Chongxi Bao, Yang Xie, and Ankur Srivastava. A security-aware design scheme for better hardware trojan detection sensitivity. In *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*, pages 52–55. IEEE, 2015.

[41] Chongxi Bao, Domenic Forte, and Ankur Srivastava. On reverse engineering-based hardware trojan detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):49–57, 2016.

[42] Chongxi Bao, Domenic Forte, and Ankur Srivastava. On application of one-class svm to reverse engineering-based hardware trojan detection. In *Quality Electronic Design (ISQED), 2014 15th International Symposium on*, pages 47–54. IEEE, 2014.

[43] Chongxi Bao and Ankur Srivastava. 3d integration: New opportunities in defense against cache-timing side-channel attacks. In *ICCD*, pages 294–301. IEEE, 2015.

[44] Chongxi Bao and Ankur Srivastava. Exploring timing side-channel attacks on path-orams. In *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*. IEEE, 2017.

[45] Yuriy Shiyanovskii, F Wolff, Aravind Rajendran, C Papachristou, D Weyer, and W Clay. Process reliability based trojans through nbti and hci effects. In *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on*, pages 215–222. IEEE, 2010.

[46] Y Jin and Y Makris. Hardware trojans in wireless cryptographic integrated circuits. *Design & Test, IEEE*, PP(99):1–1, 2013.

[47] Xuehui Zhang and M Tehranipoor. Case study: Detecting hardware trojans in third-party digital IP cores. pages 67–70, May.

[48] F. Wolff, C. Papachristou, S. Bhunia, and R.S. Chakraborty. Towards trojan-free trusted ICs: problem analysis and detection scheme. pages 1362–1365, October.

[49] RajatSubhra Chakraborty, Francis Wolff, Somnath Paul, Christos Papachristou, and Swarup Bhunia. MERO: a statistical approach for hardware trojan detection. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 396–410. Springer Berlin Heidelberg, 2009.

[50] S. Narasimhan, Xinmu Wang, Dongdong Du, R.S. Chakraborty, and S. Bhunia. TeSR: a robust temporal self-referencing approach for hardware trojan detection. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 71 –74, June 2011.

[51] S. Narasimhan, Dongdong Du, R.S. Chakraborty, S. Paul, F. Wolff, C. Papachristou, K. Roy, and S. Bhunia. Multiple-parameter side-channel analysis: A non-invasive hardware trojan detection approach. pages 13–18.

[52] Yu Liu, Ke Huang, and Yiorgos Makris. Hardware trojan detection through golden chip-free statistical side-channel fingerprinting. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014.

[53] Byeongju Cha and Sandeep K Gupta. Trojan detection via delay measurements: A new approach to select paths and vectors to maximize effectiveness and minimize cost. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 1265–1270. IEEE, 2013.

[54] Seetharam Narasimhan and Swarup Bhunia. Hardware trojan detection. In *Introduction to Hardware Security and Trust*, pages 339–364. Springer, 2012.

[55] Tianwei Zhang and Ruby B Lee. New models of cache architectures characterizing information leakage from cache side channels. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 96–105. ACM, 2014.

[56] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *S & P*, pages 191–205. IEEE, 2013.

[57] Onur Aciiçmez. Yet another microarchitectural attack:: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18. ACM, 2007.

[58] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 110–124. Springer, 2010.

[59] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers Track at the RSA Conference*, pages 225–242. Springer, 2007.

[60] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[61] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 490–505. IEEE, 2011.

[62] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer, 2014.

[63] Joop van de Pol, Nigel P Smart, and Yuval Yarom. Just a little bit more. In *Cryptographers Track at the RSA Conference*, pages 3–21. Springer, 2015.

[64] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[65] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014.

[66] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. ooh aah... just a little bit: A small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 75–92. Springer, 2014.

[67] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme. *exchange*, 6(18):24, 2016.

[68] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Workshop on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[69] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 201–215. Springer, 2006.

[70] Hassan Aly and Mohammed ElGayyar. Attacking aes using bernstein's attack on modern processors. In *AFRICACRYPT*, pages 127–139. Springer, 2013.

[71] Dan Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint Archive*, 2005:280, 2005.

[72] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[73] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.

[74] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 194–199. IEEE, 2011.

[75] Yinqian Zhang and Michael K Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 827–838. ACM, 2013.

[76] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 203–215. IEEE, 2014.

[77] Zhenghong Wang and Ruby Lee. A novel cache architecture with enhanced performance and security. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 83–93. IEEE, 2008.

[78] Jingfei Kong, Onur Aciiçmez, J-P Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 393–404. IEEE, 2009.

[79] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. volume 40, pages 118–129. IEEE Computer Society, 2012.

[80] Peng Li, Debin Gao, and Michael K Reiter. Mitigating access-driven timing channels in clouds using stopwatch. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[81] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, volume 20, page 12, 2012.

[82] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[83] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100. ACM, 2011.

[84] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Advances in Cryptology–CRYPTO 2010*, pages 502–519. Springer, 2010.

[85] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *USENIX Security 14*, pages 749–764, 2014.

[86] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. *IACR ePrint Archive*, 2014:997, 2014.

[87] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.

[88] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[89] R. Torrance and D. James. The state-of-the-art in semiconductor reverse engineering. pages 333–338, May.

[90] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. 2009.

[91] Yier Jin, Nathan Kupp, and Yiorgos Makris. Dftt: design for trojan test. In *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pages 1168–1171. IEEE, 2010.

[92] Mainak Banga and Michael S Hsiao. Vitamin: Voltage inversion technique to ascertain malicious insertions in ics. In *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*, pages 104–107. IEEE, 2009.

[93] Hassan Salmani, Mohammad Tehranipoor, and Jim 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. IEEE, 2009.

[94] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. A novel technique for improving hardware trojan detection and reducing trojan activation time. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(1):112–125, 2012.

[95] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. A layout-aware approach for improving localized switching to detect hardware trojans in integrated circuits. In *Information Forensics and Security (WIFS), 2010 IEEE International Workshop on*, pages 1–6. IEEE, 2010.

[96] Vladimir Vapnik. *The nature of statistical learning theory*. springer, 2000.

[97] Bernhard Schlkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):14431471, 2001.

[98] Nauman Shahid, Ijaz Haider Naqvi, and Saad Bin Qaisar. One-class support vector machines: analysis of outlier detection for wireless sensor networks in harsh environments. *Artificial Intelligence Review*, pages 1–49, 2013.

[99] DMJ Tax. *One-class classification: concept-learning in the absence of counter-examples*. PhD thesis, [Sl: sn], 2001.

[100] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:127:27, 2011. Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

[101] Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *Circuits and Systems, 1989., IEEE International Symposium on*, pages 1929–1934. IEEE, 1989.

[102] trust HUB.org. http://trust-hub.org/resources/benchmarks.

[103] Colin Percival. Cache missing for fun and profit, 2005.

[104] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, pages 719–732. USENIX Association, 2014.

[105] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H Loh, David McCauley, Pat Morrow, Donald W Nelson, Daniel Pantuso, et al. Die stacking (3d) microarchitecture. In *MICRO*, pages 469–479. IEEE, 2006.

[106] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2002.

[107] Tiantao Lu and Ankur Srivastava. Detailed electrical and reliability study of tapered tsvs. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*, pages 1–7. IEEE, 2013.

[108] Tiantao Lu and Ankur Srivastava. Electrical-thermal-reliability co-design for tsv-based 3d-ics. In *ASME 2015 International Technical Conference and Exhibition on Packaging and Integration of Electronic and Photonic Microsystems collocated with the ASME 2015 13th International Conference on Nanochannels, Microchannels, and Minichannels*, pages V001T09A037–V001T09A037. American Society of Mechanical Engineers, 2015.

[109] Tiantao Lu, Zhiyuan Yang, and Ankur Srivastava. Post-placement optimization for thermal-induced mechanical stress reduction. In *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pages 158–163. IEEE, 2016.

[110] Tiantao Lu and Ankur Srivastava. Modeling and layout optimization for tapered tsvs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(12):3129–3132, 2015.

[111] Yuh-Fang Tsai, Yuan Xie, Narayanan Vijaykrishnan, and Mary Jane Irwin. Three-dimensional cache design exploration using 3dcacti. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 519–524. IEEE, 2005.

[112] Gabriel H Loh. 3d-stacked memory architectures for multi-core processors. In *ISCA*, pages 453–464. IEEE Computer Society, 2008.

[113] Changkyu Kim, Doug Burger, and Stephen W Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *ACM SIGOPS Operating Systems Review*, 36(5):211–222, 2002.

[114] Guangyu Sun, Xiaoxia Wu, and Yuan Xie. Exploration of 3d stacked l2 cache design for high performance and efficient thermal control. In *ISLPED*, pages 295–298. ACM, 2009.

[115] Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, and Mahmut Kandemir. Design and management of 3d chip multiprocessors using network-in-memory. In *ISCA*, pages 130–141. ACM, 2006.

[116] Johannes Blömer and Volker Krummel. Analysis of countermeasures against access driven cache attacks on aes. In *Selected Areas in Cryptography*, pages 96–109. Springer, 2007.

[117] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[118] Tarush Jain and Tanmay Agrawal. The haswell microarchitecture–4th generation processor. *International Journal of Computer Science and Information Technologies*, 4(3):477–480, 2013.

[119] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 106–117. IEEE Computer Society, 2012.

[120] David Nassimi and Sartaj Sahni. A self-routing benes network and parallel permutation algorithms. *Computers, IEEE Transactions on*, 100(5):332–340, 1981.

[121] Aythan Avior, Tiziana Calamoneri, Shimon Even, Ami Litman, and Arnold L Rosenberg. A tight layout of the butterfly network. *Theory of Computing Systems*, 31(4):475–488, 1998.

[122] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 629–636. IEEE, 2015.

[123] Tianwei Zhang, Fangfei Liu, Si Chen, and Ruby B Lee. Side channel vulnerability metrics: the promise and the pitfalls. In *Proceedings of the 2nd*

International Workshop on Hardware and Architectural Support for Security and Privacy, page 2. ACM, 2013.

[124] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[125] Christianto C Liu, Ilya Ganusov, Martin Burtscher, and Sandip Tiwari. Bridging the processor-memory performance gap with 3d ic technology. *Design & Test of Computers, IEEE*, 22(6):556–564, 2005.

[126] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[127] Jonathan Valamehr, Mohit Tiwari, Timothy Sherwood, Ryan Kastner, Ted Huffmire, Cynthia Irvine, and Timothy Levin. Hardware assistance for trustworthy systems through 3-d integration. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 199–210. ACM, 2010.

[128] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324. ACM, 2013.

[129] Christopher W Fletchery, Ling Ren, Xiangyao Yu, Marius van Dijk, Omar Khan, and Srinivas Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *High Performance Computer Architecture (HPCA), 2014*, pages 213–224. IEEE, 2014.

[130] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGARCH Computer Architecture News*, 43(1):87–101, 2015.

[131] Christopher W Fletcher. *Ascend: An architecture for performing secure computation on encrypted data*. PhD thesis, MIT, 2013.

[132] Steven Gordon, Atsuko Miyaji, Chunhua Su, and Karin Sumongkayyothin. Analysis of path oram toward practical utilization. In *Network-Based Information Systems (NBiS), 2015*, pages 646–651. IEEE, 2015.

[133] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. Fork path: improving efficiency of oram by removing redundant memory accesses. In *MICRO*, pages 102–114. ACM, 2015.

[134] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.

[135] Yang Xie, Chongxi Bao, Caleb Serafy, Tiantao Lu, Ankur Srivastava, and Mark Tehranipoor. Security and vulnerability implications of 3d ics. *IEEE Transactions on Multi-Scale Computing Systems*, 2(2):108–122, 2016.