8-25-2017

Architectural Primitives for Secure Computation Platforms

Syed Kamran Haider
University of Connecticut - Storrs, syed.haider@uconn.edu

Architectural Primitives for Secure Computation Platforms

Syed Kamran Haider, Ph.D.

University of Connecticut, 2017

ABSTRACT

Computation outsourcing to a cloud cluster has become a common practice in today's world. Unlike traditional computing models, the cloud computing model has raised serious challenges in terms of privacy and integrity of the users' sensitive data. Depending on their respective adversarial models, modern secure processor architectures offer a promising solution to many of these challenges. However, almost all secure processor proposals dealing with computation outsourcing face two crucial security vulnerabilities that are still treated as open problems; the threat of hardware Trojans embedded inside the secure processor chip, and, the threat of privacy leakage via access patterns of the secure processor to the untrusted memory.

In order to deal with the above mentioned vulnerabilities, I propose various architectural primitives that address each of these challenges individually in an efficient manner. For hardware Trojans detection, a first rigorous algorithm HaTCh is proposed that not only detects a small set of publicly known Trojans, but also detects an exponentially large class of deterministic hardware Trojans. Oblivious RAM (ORAM) is an established technique that hides memory access patterns through redundant accesses, thereby preventing privacy leakage. However such redundancy incurs a large performance overhead. Therefore, a dynamic prefetching technique tailored to ORAM is proposed that detects and exploits data locality without leaking any information on the access pattern, and results in significant performance gain. Recent research has shown that information can still be leaked even if only the memory write-access pattern (not reads) is visible to the adversary. For such weaker adversaries, a fully functional ORAM causes unnecessary overheads. With this intuition in mind, an efficient write-only ORAM scheme has been proposed which substantially outperforms the closest existing write-only ORAM from the literature.

Architectural Primitives for Secure Computation Platforms

Syed Kamran Haider

M.S., University of Kaiserslautern, Kaiserslautern, Germany, 2013

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2017

APPROVAL PAGE

Doctor of Philosophy Dissertation

Architectural Primitives for Secure Computation Platforms

Presented by

Syed Kamran Haider, M.S.

Major Advisor _____
Marten van Dijk

Associate Advisor _____
Omer Khan

Associate Advisor _____
John Chandy

University of Connecticut

2017

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my advisor Prof. Marten van Dijk for the continuous support of my Ph.D study and related research, for his patience, and motivation. His guidance helped me throughout the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Omer Khan, and Prof. John Chandy, for their insightful comments and encouragement, but also for the critical questions which incented me to widen my research from various perspectives.

My sincere thanks also goes to Prof. Srini Devadas, and Dr. Dan Alistarh, who provided me opportunities to join their teams as a collaborator. Without their precious support it would not be possible to conduct several aspects of this research.

I also thank my fellow labmates and collaborators for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last four years.

Last but not the least, I would like to thank my family: my parents and my siblings for their unconditional support throughout my life.

Contents

List of Figures

List of Tables

Page

Chapter 1

Introduction

The high computational demands of modern digital world have resulted in an increased trend of *outsourcing* the computation to a powerful compute cluster, or a *cloud*. While facilitating the end user, computation outsourcing raises serious concerns regarding the privacy, integrity and freshness of the user's data and computation.

One solution to this problem is to use *tamper-resistant* hardware based secure processors to perform the computation. In this setting, only the processor's chip is considered *trusted*. The user sends his/her encrypted data to the trusted secure processor, inside which the data is decrypted and computed upon. The final results are encrypted and sent back to the user. The trusted processor is assumed to be tamper-resistant, namely, an external adversary is not able to look inside the chip to learn any information. Many such hardware platforms have been proposed, including TPM [1, 2, 3], Intel's TPM+TXT [4], eXecute Only Memory (XOM) [5, 6, 7], Aegis [8, 9], Ascend [10], Phantom [11] and Intel's SGX [12].

Although depending upon their specific adversarial model, the trusted hardware based secure processors provide several different security guarantees (c.f. Figure 1.0.1). However, the following two adversarial capabilities are applicable to almost all adversarial models dealing with computation outsourcing, and can result in security attacks in nearly all secure processor implementations. First, a Hardware Trojan embedded inside the secure processor chip; and second, privacy leakage via data dependent access patterns to untrusted

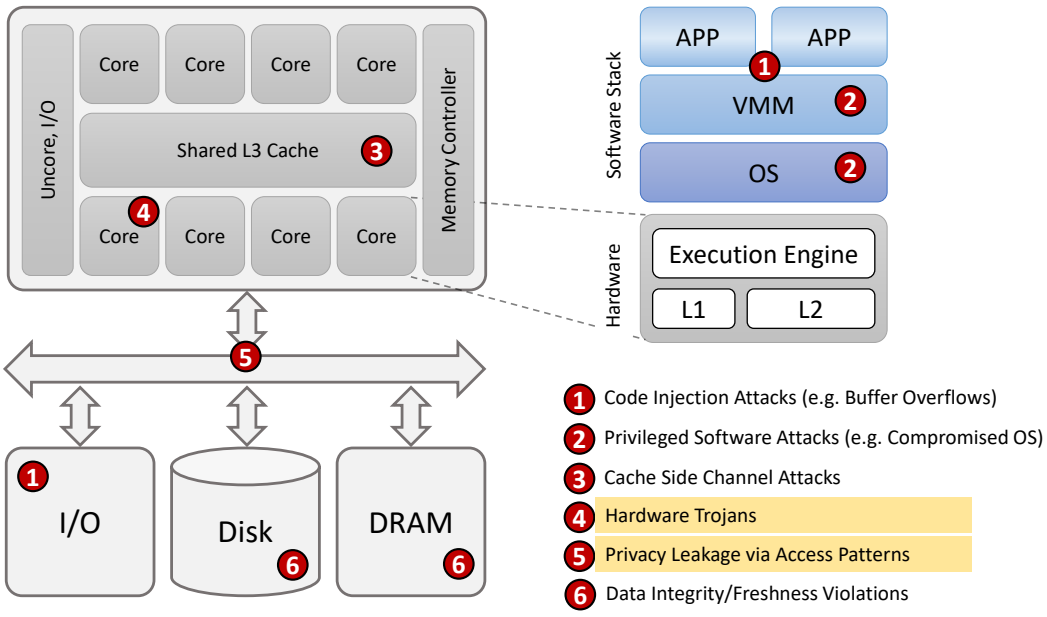


Figure 1.0.1: Attack surface of different security attacks on a computer system. Attack ④ and ⑤ (highlighted in yellow) are the only ones considered in the current adversarial model.

storage (Figure 1.0.1, Attack ④ and ⑤ respectively).

Hardware designers frequently use third party IP cores as black boxes instead of building these logic blocks from scratch, in order to save the valuable time and other resources. However, these third party IP cores can contain Hardware Trojans (HTs) which could potentially harm the normal functionality of the secure processor (i.e. denial of service attack) or cause privacy leakage [13, 14, 15, 16]. These Trojans must be detected in *pre-silicon* phase in order to prevent an adversary from infecting millions of secure processor chips through a Trojan affected IP core. Over the past decade, Hardware Trojans (HTs) research community has made significant progress towards developing effective countermeasures for various types of HTs, yet these countermeasures are shown to be circumvented by sophisticated HTs designed subsequently. Therefore, instead of guaranteeing a certain (low) false negative rate for a small *constant* set of publicly known HTs, a rigorous security framework of HTs should provide an effective algorithm to detect any HT from an *exponentially large* class (exponential in number of wires in IP core) of HTs with negligible false negative rate.

Since a (secure) processor has limited on-chip memory, it cannot hold all of the user's data inside the trusted boundary. Consequently, the data must be stored in the untrusted memory outside the secure processor's trusted boundary. While an adversary cannot learn about user's data residing inside the trusted boundary, information can still be leaked through accesses to the untrusted main memory. Although all the data stored in the external memory can be encrypted to hide the data values, the memory access pattern (i.e., address sequence) may leak information [17]. Completely preventing leakage from the memory access pattern requires the use of Oblivious RAM (ORAM) [18]. ORAM translates a single read/write operation into several accesses to randomized locations. As a result, the locations touched in each ORAM read/write would have exactly the same distribution and be indistinguishable to an adversary.

The cost of ORAM security is performance. Each ORAM access needs to touch multiple physical locations which incurs one to two orders of magnitude more bandwidth and latency when compared to a normal DRAM. Path ORAM [19], the most efficient and practical ORAM system for secure processors so far, still incurs at least 30× more latency than a normal DRAM for a single access. This results in $2-10\times$ performance slowdown [10, 20]. It is of utmost importance to devise novel methodologies addressing the performance bottlenecks of ORAM in order to design efficient secure processors.

1.1 Proposed Research Directions

1.1.1 Hardware Trojans Detection

Although Hardware Trojans (HT) detection community has done tremendous efforts to develop effective countermeasures to detect several kinds of HTs, however still, the HT design research has always been one step ahead of it. State of the art HT detection schemes namely Unused Circuit Identification (UCI) [21], VeriTrust [22], and FANCI [23] have shown that they can detect all HTs from the TrustHub [24] benchmark suite. On the other hand, researchers have also shown that these schemes can still be circumvented by carefully designing new HTs [25, 26]. A recent work, DeTrust [27], in fact introduces a systematic methodology of designing Trojans that can evade both VeriTrust and FANCI.

The fundamental reason behind the possibility of circumventing these HT detection techniques is the lack of rigorous analysis about the coverage of these techniques at their design time. Typically, HT detection techniques offer guaranteed coverage for a small *constant* set of *publicly known* HT benchmarks such as TrustHub. However, later on, one may design a HT which is slightly different than previously known HTs such that it can bypass the detection schemes. Clearly the HT detection techniques must target and provide security guarantees against certain behavioral traits resulting in a larger *class* of HTs instead of targeting specific known HTs.

Therefore, I first propose a formal framework for trigger activated deterministic HTs based on their several advanced properties, which include *trigger signal dimension*, *payload propagation delay*, *implicit malicious behavior factor*, and *trigger signal locality*. This framework introduces the design principles that an adversary might follow to come up with sophisticated HTs in order to bypass the current HT countermeasures, and as a result it characterizes a large group H_D of HTs. The aforementioned group H_D covers vast ground on the HT landscape including all known trigger based deterministic HTs, and is therefore a good candidate for a countermeasure to target.

I then propose a powerful HT detection algorithm called ***Ha**rdware **T**rojan **C**atc**h**er* (HaTCh) that offers complete coverage and transparent security guarantees for the above mentioned HT class H_D. The proposed scheme takes into account the advanced properties of the HTs described above to provide complete coverage over the class H_D. HaTCh works in two phases; a *learning phase* and a *tagging phase*. The learning phase performs logic testing on the IP core and produces a blacklist of all the unactivated (and potentially untrusted) transitions of internal wires. The tagging phase adds additional logic to the IP core to track these untrusted transitions. If these untrusted transitions – that are potentially related to a HT trigger circuit – ever occur in the lifetime of the circuit, an exception is raised that shows the activation of a HT.

Although VeriTrust and FANCI can detect publicly known TrustHub HTs, it is unclear what security guarantees they offer outside TrustHub. HaTCh, on the other hand, is capable of detecting any HT from H_D, whether the HT is publicly known or unknown. HaTCh is proven to offer a negligible false negative rate $\leq 2^{-\lambda}$ and a controllable false positive rate $\leq \rho$ for user defined parameters λ and ρ.

1.1.2 Efficient Fully-Functional ORAM

Path ORAM [19] is currently the most efficient ORAM scheme for limited client (processor) storage, and, further, is appealing due to its simplicity. It has two main hardware components: the *binary tree storage* and the *ORAM controller*. Binary tree stores the data content of the ORAM in its nodes, and is implemented on DRAM. ORAM controller is a piece of trusted hardware that controls the tree structure. Besides necessary logic circuits, the ORAM controller contains two main structures, a *position map* and a *stash*. The *position map* is a lookup table that associates the program address of a data block (a) with a path in the ORAM tree (path s). The *stash* is a piece of memory that stores up to a small number of data blocks at a time. Whenever block a is requested by the processor, the ORAM controller reads (and writes back) the whole path s at which the data block a resides.

Clearly, the access latency is the main bottleneck in Path ORAM, a natural solution that comes to mind is to apply latency hiding techniques, such as "data prefetching". Traditional prefetching is helpful for the (insecure) DRAM, however it is not the case for ORAM. The much longer latency (more than 30×) and lower throughput (2 orders of magnitude) of ORAM leads to two effects. First, it is not useful to overlap multiple ORAM accesses, since a single ORAM access already fully utilizes the entire DRAM bandwidth. Second, for memory bound applications, ORAM requests line up in the ORAM controller and there is no idle time for prefetching. In other words, traditional prefetching is likely to block normal requests and hurt performance.

For prefetching under ORAM, a notion of *super block* was first proposed in [20] which tries to exploit spatial locality in Path ORAM. In particular, it statically groups every n consecutive blocks in the address space as a *super block* and maps all the blocks belonging to a super block to the same path. Whenever one block in a super block is accessed, all the blocks in that super block are loaded from the path and remapped to a same random path. The block of interest is returned to the processor and the other blocks are *prefetched* and put into the LLC (Last Level Cache). The idea is that the prefetched blocks may be accessed in the near future due to spatial data locality, which saves some expensive Path ORAM accesses. Although the static super block scheme provides performance gain for programs with good locality, it has significant limitations which make it not practical. First, it significantly hurts performance when the program has bad

spatial locality. With these programs, prefetching always misses and pollutes the cache. Second, it cannot adjust to different program behaviors in different phases. In practice, this leads to suboptimal performance for certain programs. Third, it is the responsibility of programmers or the compiler to figure out whether the super block scheme should be used and the size of the super block.

In order to address the limitations of the static super block scheme, I propose a dynamic ORAM prefetching scheme called *dynamic super block*. The proposed scheme essentially groups (*merge* operation) or ungroups (*break* operation) the blocks dynamically depending upon the program behavior. Following are the key properties of dynamic super block scheme: (1) Crucially, super block merging is determined at runtime. Only the blocks that exhibit spatial locality are merged into super blocks. Programmers or compilers are not involved in this process. (2) In determining whether blocks should be merged into a super block, the dynamic super block scheme also takes into account the *ORAM access rate*, *prefetch hit rate*, etc. For example, if the prefetch hit rate is too low, merging should be stopped. (3) Finally, when a super block stops showing locality, the super block is broken. This makes it possible to adjust to program phases.

1.1.3 Efficient Write-Only ORAM

A key observation regarding the adversarial model assumed by Path ORAM is that the adversary is capable of learning fine-grained information of *all* the accesses made to the memory. This includes the information about which location is accessed, the type of operations (read/write), and the time of the access. It is an extremely strong adversarial model which, in most cases, requires direct physical access to the memory address bus in order to monitor both read *and* write accesses, e.g. the case of a *curious* cloud server.

On the other hand, for purely remote adversaries (where the cloud server itself is trusted), direct physical access to the memory address bus is not possible thereby preventing them from directly monitoring read-/write access patterns. Such remote adversaries, although weaker than adversaries having physical access, can potentially still "learn" the application's write access patterns. Interestingly, privacy leakage is still possible even if the adversary is able to infer just the write access patterns of an application. John *et al.* demonstrated such an attack [28] on the famous Montgomery's ladder technique [29] commonly used for modular exponentiation in public key cryptography.

Although Path ORAM also offers a solution to such weaker adversaries. However, the added protection (obfuscation of reads) offered by Path ORAM is unnecessary and is practically an overkill in this scenario which results in significant performance penalty. A far more efficient solution to this problem would be a *write-only* ORAM scheme, which only obfuscates the write accesses made to the memory. Since read accesses leave no trace in the memory and hence do not need to be obfuscated in this model, a write-only ORAM can offer significant performance advantage over a fully functional ORAM.

Based on the above mentioned insights, I present an efficient write-only ORAM implementation, called *Flat ORAM*, tailored specifically for secure processors. The closest related work in the literature, HIVE [30], proposes a write-only ORAM scheme for implementing hidden volumes in hard disk drives. However, HIVE suffers from performance bottlenecks while managing the memory occupancy information vital for correctness of the protocol. Flat ORAM architecture resolves these bottlenecks by introducing a crucial data structure, called Occupancy Map, which efficiently manages the memory occupancy information resulting in far better performance. However, with the introduction of occupancy map structure, several challenges arise with respect to securely storing and managing the occupancy map and accessing it without leaking privacy. The proposed scheme efficiently tackles these challenges. Furthermore, it seamlessly adopts the basic structures of Path ORAM (e.g. Position Map, Stash) as well as various crucial optimizations proposed over the past decade. Simulation results show that, on average, Flat ORAM offers up to 50% performance gain and 80% energy savings over HIVE.

1.2 Dissertation Contributions

The following key contributions have been made by this dissertation:

1. The first ever rigorous Hardware Trojan detection algorithm, HaTCh, that detects any Trojan from H_D, a huge class of deterministic Trojans which is orders of magnitude larger than the small subclass (e.g. TrustHub) considered in the current literature.

2. A dynamic ORAM prefetching technique that detects data locality in programs at runtime, and ex-

ploits the locality without leaking any information on the access pattern. It improves Path ORAM performance by 20.2% (upto 42.1%) for memory bound Splash2 benchmarks [31], 5.5% for SPEC06 benchmarks [32], and 23.6% and 5% for YCSB and TPCC in DBMS benchmarks [33] respectively.

3. An efficient write-only ORAM scheme for secure processors that avoids high performance penalties of Path ORAM for weaker adversarial models, as well as outperforms the closest write-only schemes (e.g., HIVE) existing in the current literature. On average, Flat ORAM only incurs a moderate slowdown of 3× over the insecure DRAM for memory intensive benchmarks among Splash2 and 1.6× for SPEC06. Compared to HIVE, Flat ORAM offers 50% performance gain on average and up to 80% energy savings.

Chapter 2

Advancing the State-of-the-Art in Hardware Trojans Design

Modern electronic systems heavily use third party IP cores as their basic building blocks. These IP cores give rise to a critical security problem: how to make sure that the IP core does not contain a *Hardware Trojan* (HT)? A significant amount of research has been done during the past decade to design efficient tools for HT detection. Hicks *et al.* [21] proposed *Unused Circuit Identification* (UCI) which centers on the fact that the HT circuitry mostly remains inactive within a design, and hence such minimally used logic can be distinguished from the other parts of the circuit. However later works [25], [26] showed how to design HTs which can defeat the UCI detection scheme. Zhang *et al.* [22] and Waksman *et al.* [23] proposed detection schemes called VeriTrust and FANCI respectively and showed that they can detect all HTs from the TrustHub [24] benchmark suite. Yet again, a more recent technique called DeTrust [27] introduces new Trojan designs which can evade both VeriTrust and FANCI.

The reason behind this cat-and-mouse game between attackers and defenders is that the current HT detection tools offer critically low HT coverage and typically only cover a small *constant* set of *publicly known* HT benchmarks such as TrustHub. Whereas an adversary may design new HTs which are different

This work has been published at *IEEE International Midwest Symposium on Circuits and Systems, 2017* [34].

from the publicly known HTs in that they can bypass the detection tool, as demonstrated by DeTrust [27]. It is unclear to what extent the existing HT detection tools are effective for *publicly unknown* HTs. This work, therefore, stresses that instead of guaranteeing a certain (low) false negative rate for a small *constant* set of publicly known HTs, a rigorous HT detection tool should guarantee the detection of an *exponentially large* class (exponential in number of wires in IP core) of HTs with negligible false negative rate. In this work, we limit ourselves to trigger activated digital HTs which have digital payloads. HTs that are always active and/or exploit side channels for their payloads are out of scope of this work.

This chapter extends the design space of trigger activated and deterministic digital HTs by introducing their four crucial properties (d, t, α, l) which characterize a large and complex class H_D. These properties, determining the stealthiness of HTs, lead to a much more detailed classification of such HTs and hence assign well defined boundaries to the scope of the existing and new countermeasures on the huge landscape of HTs. In our model, H_D represents the HTs which are embedded in a digital IP core whose output is a function of only its input, and the algorithmic specification of the IP core can exactly predict the IP core behavior. A brief highlight of the properties of H_D is as follows:

Trigger Signal Dimension d represents the number of wires used by HT trigger circuitry to activate the payload circuitry in order to exhibit malicious behavior. Large d shows a complicated trigger signal, hence harder to detect.

Payload Propagation Delay t is the number of cycles required to propagate malicious behavior to the output port *after* the HT is triggered. Large t means it takes a long time after triggering until the malicious behavior is seen, hence less likely to be detected during testing.

Implicit Behavior Factor α represents the probability that given a HT gets triggered, it will not (explicitly) manifest malicious behavior; this behavior is termed as implicit malicious behavior. Higher probability of implicit malicious behavior means higher stealthiness.

Trigger Signal Locality l shows the spread of trigger signal wires of the HT across the IP core. Large l means that these wires are spread out in the circuit, hence the HT is harder to detect.

Based on this framework, we discover that the current publicly known HTs have small d (mostly $d = 1$

for TrustHub[1]) and hence they are the simplest ones. Although VeriTrust and FANCI can detect TrustHub HTs, it is unclear what security guarantees they offer outside TrustHub. This framework allows us to design new stealthy HTs which cannot be efficiently detected by ordinary means (design knowledge of the HT itself needs to be incorporated in the detection tool).

2.1 Characterization of Hardware Trojans

A Hardware Trojan (HT) is malicious *extra* circuitry embedded inside a larger circuit, which results in data leakage or harm to the normal functionality of the circuit once activated. We define *extra* circuitry as redundant logic added to the IP core without which the core can still meet its design specifications[2]. A *trigger activated* HT activates upon some special event, whereas an *always active* HT remains active all the time to deliver the intended payload.

Trigger activated HTs typically consist of two parts: a *trigger circuitry* and a *payload circuitry*. The trigger circuitry is implemented semantically as a comparator which compares the value(s) of certain wires(s) of the circuit with a specified boolean value called *trigger condition*. The HT trigger circuitry sends its comparator's output to the payload circuitry over certain other wire(s) called the *trigger signal*. Once the trigger signal is asserted, the payload circuitry performs the malicious operation called 'payload' as intended by the adversary. The trigger signal must not be confused with trigger condition; trigger condition is an event which causes the HT activation, whereas trigger signal is the output of trigger circuitry which tells the payload circuitry to show malicious behavior.

We first characterize the trigger activated HTs based on the payload channels they use once activated, as shown in Figure 2.1.1. *St* refers to the Trojans using only standard I/O channels, whereas *Si* represents the Trojans which also use side channels to deliver the payload. If a Trojan delivers some of its payload over the timing channel (or other side channels), then we define it to be in *Si*. If a Trojan delivers *all* of its payload using the standard usage of I/O channels, then we define it to be in *St*.

[1] Only trigger-activated digital HTs are considered from TrustHub.

[2] Design specifications can also cover the performance requirements of the core, and hence pipeline registers etc. added to the core only for performance reasons can also be considered as 'necessary' to meet the design specifications and will not be counted towards 'extra' circuitry.

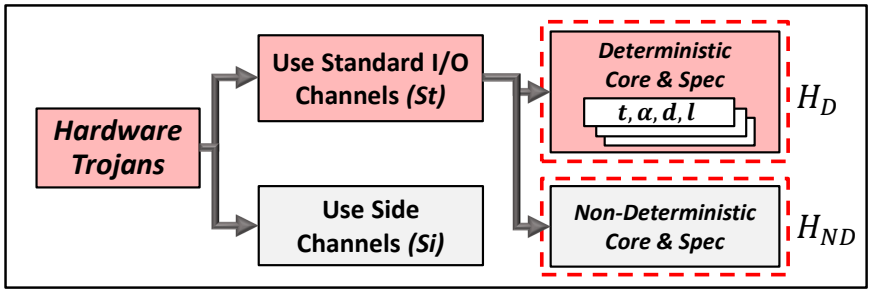


Figure 2.1.1: Characterization of Hardware Trojans

We further refine our description of *St* Trojans by subdividing them in H_D and H_{ND} groups based on the IP core behavior in which they are embedded and their algorithmic specifications. H_D Trojans are the ones which are: (1) Embedded in an IP core whose output is a function of only its input – i.e. the logical functionality of the IP core is deterministic, and, (2) The algorithmic specification of the IP core can exactly predict the IP core behavior. If any of the two above mentioned conditions is not satisfied for a *St* type HT then we consider the HT to be in H_{ND}.

A true random number generator (TRNG), for example, is a non-deterministic IP core since its output cannot be predicted and verified by logic testing against an expected output.[3] Any *St* HT in such a core is considered H_{ND}. A pseudo random number generator (PRNG), on the other hand, is considered a deterministic IP core as its output depends upon the initial seed and is therefore predictable (hence H_D). Similarly, if the algorithmic specification allows coin flips generated by a TRNG then we consider the HT to be H_{ND}, whereas if the coin flips are from a PRNG then we regard the HT as H_D.

The non-deterministic behavior of IP cores and/or their functional specification which accepts small probabilistic fluctuations within some acceptable range allows a covert channel for H_{ND} Trojans to embed some minimal malicious payload in the standard output without being detected by an external observer [35]. The external observer considers these small fluctuations as part of the functional specification. Hence, the non-deterministic nature of H_{ND} Trojans prohibits the development of a logic testing based tool to detect these Trojans with overwhelming probability.

[3] Any IP core which contains a TRNG as a module, yet the I/O behavior of the core can still be predicted is considered H_D.

2.2 Advanced Properties of Class H_D

Figure 2.2.1 shows a simple H_D Trojan embedded in a half adder circuit. The HT-free circuit in Figure 2.2.1a generates a sum $S = A \oplus B$ and a carry $C = A \cdot B$. The HT, highlighted in red in Figure 2.2.1b, triggers when $A = B$ and produces incorrect results i.e. $S = B$ for $A = B$ and $S = A \oplus B$ for $A \neq B$. Notice the difference between the trigger condition $A = B$, and the trigger signal *Sel* which only becomes 1 when the trigger condition is satisfied.

The Trojan affected circuit in Figure 2.2.1 produces a *malicious* output $S = 1$ for trigger condition $A = B = 1$ which is distinguishable from otherwise normal output ($S = 0$). However, the same circuit produces a (so called) malicious output $S = 0$ for trigger condition $A = B = 0$ which is the same as otherwise normal output and cannot be distinguished from the 'normal' behavior of the circuit. This observation leads us to the definition of *explicit* vs. *implicit* malicious behaviors:

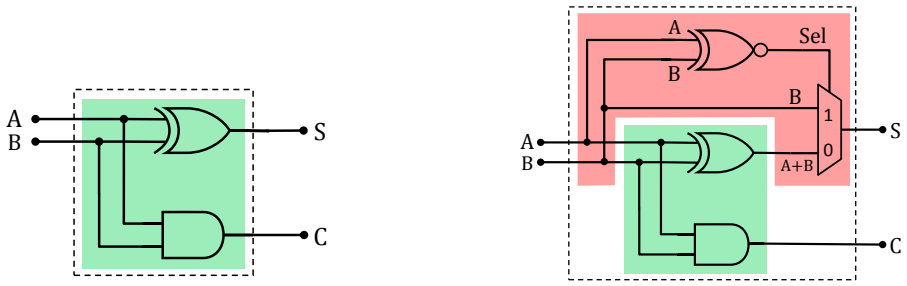
Definition 2.2.1. Explicit malicious behavior refers to a behavior of a HT where the HT generated output is *distinguishable* from a normal output.

Definition 2.2.2. Implicit malicious behavior refers to a behavior of a HT where the HT generated output is *indistinguishable* from a normal output.

Notice that an adversary may exploit the implicit malicious behavior to bypass the functional testing phase of the detection tools. Once the infected circuit has passed the testing and is deployed, it can then manifest explicit malicious behavior to actually deliver the payload. In other words, implicit malicious behavior can lead to false negatives. In the following discussion, we introduce some crucial properties of H_D Trojans that characterize their complexity and stealthiness, and explain them using the above mentioned example.

2.2.1 Trigger Signal Dimension (d)

When a trigger condition of a HT occurs, *regardless of the other subsequent user interactions*, its trigger circuitry gets activated and outputs a certain binary value on a certain trigger signal *Trig* to activate the



(a) Half Adder (b) Half Adder with HT

Figure 2.2.1: A simple HT: Trigger condition $A = B$; Normal output $S = A \oplus B$; Malicious output (under trigger condition) $S = B$.

payload circuitry which manifests malicious behavior. A trigger signal $Trig$ is represented as a labeled binary vector of one or more wires/registers/flip-flops (each carrying a 0 or 1), e.g. in Figure 2.2.1, $Sel = 1$ is a trigger signal. In other words, $Trig$ represents a trigger state of the circuit through which the circuit must have passed before manifesting malicious behavior. Hence a HT can be represented by a set of trigger states $\mathcal{T}$; i.e. the states which always lead to malicious behavior (implicit or explicit).

Definition 2.2.3. A set $\mathcal{T}$ of trigger states *represents* a HT if the HT always passes through one of the states in $\mathcal{T}$ in order to express implicit or explicit malicious behavior.

Notice that $\mathcal{T}$ is not unique. For example, $\mathcal{T}$ could be $\mathcal{T} = \{(Sel = 1)\}$, or it could also be $\mathcal{T} = \{[(A,B) = (1,1)], [(A,B) = (0,0)]\}$ etc. We define the *trigger signal dimension* d of a HT represented by a set of trigger states $\mathcal{T}$ as follows:

Definition 2.2.4. Trigger Signal Dimension $d(\mathcal{T})$ of a HT is defined as $d(\mathcal{T}) = \max_{Trig \in \mathcal{T}} |Trig|$.

In other words, d shows the width of the widest trigger signal bit-vector of a HT. E.g. for Figure 2.2.1b, $d(\mathcal{T}) = 1$ since the trigger signal *Sel* is only 1-bit wide (and hence easy to detect). Obviously, it becomes difficult to detect HTs which only have high dimensional sets of trigger states, i.e. multi-bit trigger signals. The set of possible values of a given trigger signal $Trig$ grows exponentially in $d = |Trig|$ and only one value out of this set can be related to the occurrence of the corresponding trigger condition. Clearly, since in theory d can be as large as the number of wires n in the IP core, H_D represents an exponentially (in n) large

class of possible HTs.

2.2.2 Payload Propagation Delay (t)

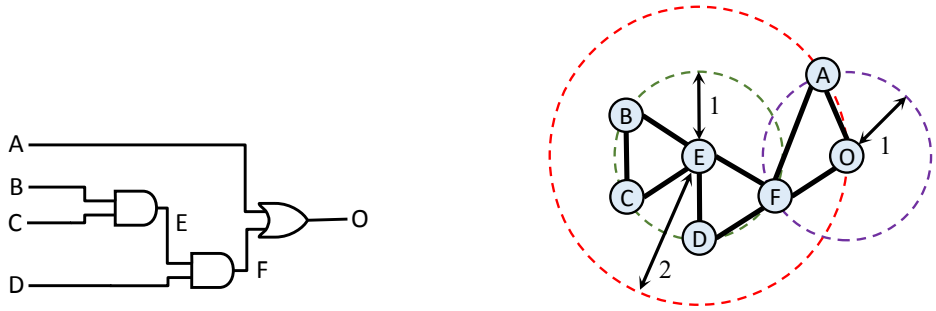
For a set $\mathcal{T}$ of trigger states, we know that if the HT manifests malicious behavior, then it must have transitioned through a trigger state $Trig \in \mathcal{T}$ at some previous clock cycle. Therefore, we define the *payload propagation delay t* as follows:

Definition 2.2.5. Payload Propagation Delay $t(\mathcal{T})$ of a HT represented by a set of trigger states $\mathcal{T}$ is defined as the *maximum* number of clock cycles taken to propagate the malicious behavior to the output *after* entering a trigger state in $\mathcal{T}$.

I.e., the number of clock cycles from the moment when a trigger signal is asserted till its resulting malicious behavior shows up at the output port. E.g., consider a counter-based HT where malicious behavior immediately (during the same clock cycle) appears at the output as soon as a counter reaches a specific value. Then, $t(\{Trig\}) = 0$ for the trigger signal $Trig$ which represents the occurrence of the specific counter value, say X (i.e. the trigger condition). However, notice that any counter value j clock cycles before X can also be considered as a trigger signal $Trig$ with $t(\{Trig\}) = j$, because eventually after j cycles this $Trig$ manifests the malicious behavior. To detect a HT with a large value of $t(\mathcal{T})$, the memory requirement and complexity of logic testing based detection tool is increased. However, for any HT, typically there exists a set $\mathcal{T}$ having a small t because of a small number of register(s) between the trigger signal and the output port.

2.2.3 Implicit Behavior Factor (α)

In addition to previously discussed properties of Trojans, the IP core in which the Trojan is embedded plays a critical role in its stealthiness. According to the definition of implicit malicious behavior, it may not always be possible to distinguish a malicious output from a normal output just by monitoring the output ports. Consequently, the implicit malicious behavior adds to the stealthiness of the HT since it creates a possibility of having a false negative under logic testing based techniques. We quantify this possibility by defining the *implicit behavior factor* α as follows:



(a) A simple circuit

(b) Locality Graph of the circuit

Figure 2.2.2: Example of Locality Graph

Definition 2.2.6. Implicit Behavior Factor $\alpha(\mathcal{T})$ of a HT is the probability that given the HT is triggered, it will manifest *implicit* malicious behavior.

In other words, the higher the value of α, the lower the chance of detection by logic testing even if the HT gets triggered and hence the higher the overall stealthiness of the HT. E.g. for the HT from Figure 2.2.1b with the trigger condition $A = B$, if $(A,B) = (1,1)$ then the malicious output $S = 1$ is distinguishable from the normal output (i.e. $S = 0$). However for $(A,B) = (0,0)$, the malicious[4] output $S = 0$ is indistinguishable from the normal output (i.e. $S = 0$), i.e. the implicit malicious behavior comes into the picture. Hence, given that this particular HT activates, the probability that the HT-generated (malicious) output is indistinguishable from the normal output is $\alpha(\mathcal{T}) = 0.5$ which represents the implicit behavior factor of this HT.

2.2.4 Trigger Signal Locality (l)

We notice that the individual wires of a HT trigger signal of dimension d are located in the close vicinity of each other in the circuit netlist/layout. This is because eventually these wires need to coordinate (through some combinational logic) with each other to perform the malicious operation. Based on this observation, we introduce the idea of *locality* in gate level circuits, similar to the region based approach in [36].

Consider the simple combinational circuit from Figure 2.2.2a for which we draw a *locality graph* shown in Figure 2.2.2b. The nodes of the graph represent the wires of the circuit and each edge between any two

[4]The output is considered malicious because it is generated by an activated hardware Trojan.

nodes represents connectivity of the corresponding two wires through a combinational logic level. In other words, each logic gate of the circuit is replaced by multiple edges (three in this case) in the graph which connect together the nodes corresponding to its inputs and the output. For any two nodes (i.e. wires) i and j in a locality graph, we define $dist(i, j)$ as the shortest distance between i and j. Hence $dist(i, j)$ represents the minimum number of basic combinational or sequential logic levels (e.g. logic gates and/or flip flops) between wires i and j. E.g. $dist(E,B) = dist(E,C) = 1$, whereas $dist(E,O) = dist(E,A) = 2$.

Definition 2.2.7. Trigger Signal Locality $l(\mathcal{T})$ of a HT represented by the set of trigger states $\mathcal{T}$ is defined as:

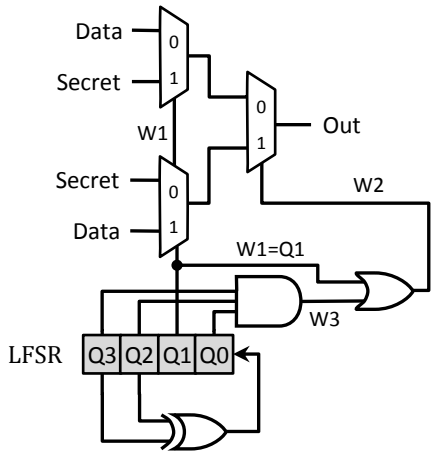
$$l(\mathcal{T}) = \max_{Trig \in \mathcal{T}} \left(\max_{0 \leq i,j < |Trig|} dist(Trig[i], Trig[j]) \right)$$

where $Trig[i]$ represents the label of the i_{th} wire in $Trig$.

A low value of $l(\mathcal{T})$ shows that the trigger signal wires of the HT are in the close vicinity of each other and vice versa. Having a notion of locality can significantly reduce the computational complexity of logic testing based HT detection tools.

2.2.5 Achievable Quadruples (d, t, α, l)

A Hardware Trojan can be represented by multiple sets of trigger states $\mathcal{T}$, each having their own d, t, α, and l values. The collection of corresponding quadruples (d,t,α,l) is defined as the achievable region of the Hardware Trojan. The choice of parameters d and t significantly affects α of the HT. α as a function of t and d is decreasing in both t and d. Reducing t means that explicit malicious behavior may not have had the chance to occur, hence, the probability α that no explicit malicious behavior is seen increases. Similarly, reducing d can increase α since as a result of smaller d, there may not exist a set of trigger signals $\mathcal{T}$ that represents the HT and satisfies $d(\mathcal{T}) \leq d$. Increasing t or d only decreases α down to a certain level; the remaining component of α represents the inherent implicit malicious behavior of the HT.



(a) A counter based H_D Trojan. Trigger Condition: $LFSR = 1101$; Trigger Signal: $(W1, W2)$; Payload: Leak *Secret*

Cycle	$Q3$	$Q2$	$Q1$	$Q0$	$W1$	$W2$	$W3$
0	1	0	1	0	1	1	0
1	0	1	0	1	0	0	0
2	1	0	1	1	1	1	0
3	0	1	1	1	1	1	0
4	1	1	1	1	1	1	1
5	1	1	1	0	1	1	0
6	1	1	0	0	0	0	0
7	1	0	0	0	0	0	0
8	0	0	0	1	0	0	0
9	0	0	1	0	1	1	0
10	0	1	0	0	0	0	0
11	1	0	0	1	0	0	0
12	0	0	1	1	1	1	0
13	0	1	1	0	1	1	0
14	**1**	**1**	**0**	**1**	**0**	**1**	**1**

(b) Truth Table of the Trojan affected circuit. Trigger Condition: $LFSR = 1101$

Figure 2.3.1: A Counter-Based H_D Trojan having Trigger Signal Dimension $d = 2$.

2.3 Examples of H_D Trojans

In this section, we present some examples of new HTs from the class H_D. These HTs demonstrate how the properties of H_D Trojans defined in the previous section play a significant role in determining the stealthiness of these HTs.

2.3.1 A Counter-Based H_D Trojan

The example Trojan shown in Figure 2.3.1a can leak *Secret* via *Out* port instead of *Data* once it is activated. The trigger condition of this Trojan is generated by a counter, when reached to (1101), which is implemented as a 4-bit maximal LFSR in order to have maximum possible time before the Trojan gets triggered. The LFSR is initialized to $(Q3, Q2, Q1, Q0) = (1010)$. At $14th$ clock cycle, the trigger condition (i.e. $LFSR = 1101$) occurs, the HT gets triggered and produces $W1 \neq W2$ on the trigger signal $(W1, W2)$. This results in

the activation of payload circuitry (the group of three MUXes) which leaks the secret.

It can be seen in Figure 2.3.1b that all individual wires related to the HT circuitry are continuously showing transitions *without* activating the HT during several clock cycles before reaching to the trigger condition (i.e. cycle 14). Therefore, clearly those existing logic testing based HT detection techniques which only look for simplistic one-dimensional HT (i.e. $d = 1$) trigger signals do not see any 'suspicious' wire which is stuck at 0 or 1, and hence this HT is not detected unless it gets activated in a long testing phase. This shows that this HT has a trigger signal dimension $d = 2$ since two wires $W1$ and $W2$ together constitute the trigger signal. Since counter based HTs can have large counters, it may not always be feasible to activate them during testing. Consequently, in order to efficiently detect such HTs, the knowledge of such design parameters must be incorporated while designing the HT countermeasures.

2.3.2 An Advanced H_D Trojan: *k-XOR-LFSR*

Figure 2.3.2 depicts *k-XOR-LFSR*, a counter based Trojan with the counter implemented as an LFSR of size k. The Trojan is merged with the circuitry of an IP core which outputs the XOR of k inputs A_j.

Let $\mathbf{r}^i \in \{0,1\}^k$ denote the LFSR register content at clock cycle i represented as a binary vector of length k. Suppose that u is the maximum index for which the linear space L generated by vectors $\mathbf{r}^0, \ldots, \mathbf{r}^{u-1}$ (modulo 2) has dimension $k-1$. Since $dim(L) = k-1 < k = dim(\{0,1\}^k)$, there exists a vector $\mathbf{v} \in \{0,1\}^k$ such that, (1) the inner products $\langle \mathbf{v}, \mathbf{r}^i \rangle = 0$ (modulo 2) for all $0 \leq i \leq u-1$, and (2) $\langle \mathbf{v}, \mathbf{r}^u \rangle = 1$ (modulo 2). *Only* the register cells corresponding to $\mathbf{v}_j = 1$ are being XORed with inputs A_j.

Since the A_j are all XORed together in the specified logical functionality to produce the sum $\sum_j A_j$, the Trojan changes this sum to

$$\sum_j A_j \oplus \sum_{j:\mathbf{v}_j=1} \mathbf{r}^i_j = \sum_j A_j \oplus \langle \mathbf{v}, \mathbf{r}^i \rangle.$$

I.e., the sum remains unchanged until the u-th clock cycle when it is maliciously inverted.

The trojan uses an LFSR to generate register values $\mathbf{r}^i \in \{0,1\}^k$ for each clock cycle i and we assume in our analysis that all vectors $\mathbf{r}^i$ behave like random vectors from a uniform distribution. Then, it is unlikely that u is more than a small constant larger than k (since every new vector $\mathbf{r}^i$ has at least probability $1/2$ to

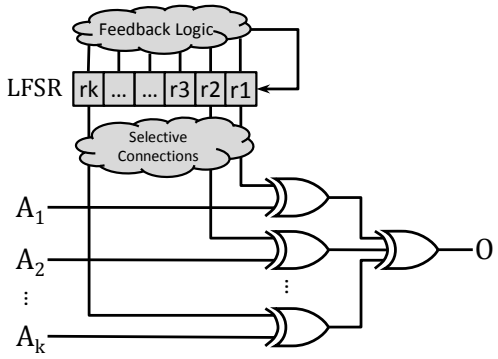


Figure 2.3.2: *k-XOR-LFSR*: A general H_D Trojan.

increase the dimension by one). Therefore, $u \approx k$, hence, the register size of the trojan is comparable to the number of clock cycles before the trojan is triggered to deliver its malicious payload. This makes the trojan somewhat contrived (since it can possibly be detected by its suspiciously large area overhead).

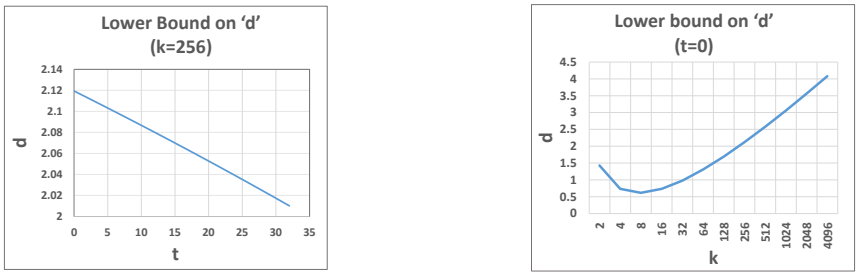
Since inputs A_j can take on any values, any trigger signal $Trig$ must represent a subset of the LFSR register content. Suppose $t(\{Trig\}) = j$. Then $Trig$ must represent a subset of $\mathbf{r}^{u-j}$. We will proceed with showing a lower bound on $d(\{Trig\})$. Consider a projection P to a subset of d register cells; by $\mathbf{r}|P$ we denote the projection of $\mathbf{r}$ under P, and we call P d-dimensional. If $\mathbf{r}^{u-j}|P \in \{\mathbf{r}^i|P : 0 \leq i < u-j\}$, then the wire combination of the d wires corresponding to $\mathbf{r}^{u-j}|P$ *cannot* represent $Trig$ (otherwise $t(\{Trig\}) > j$): if this is the case for all d dimensional P, then $Trig$ cannot represent a subset of $\mathbf{r}^{u-j}$. The probability that $\mathbf{r}^{u-j}|P \in \{\mathbf{r}^i|P : 0 \leq i < u-j\}$ is at least equal to the probability that $\{\mathbf{r}^i|P : 0 \leq i < u-j\} = \{0,1\}^d$, which is (by the union bound)

$$\geq 1 - \sum_{w \in \{0,1\}^d} Prob(\{\mathbf{r}^i|P : 0 \leq i < u-j\} \subseteq \{0,1\}^d \setminus \{w\})$$

$$= 1 - \sum_{w \in \{0,1\}^d} (1 - 1/2^d)^{u-j} \approx 1 - 2^d e^{-(u-j)/2^d}$$

Since there are $\binom{k}{d} \leq k^d/d!$ projections, $Trig$ cannot represent a subset of $\mathbf{r}^{u-j}$ with probability

$$\geq (1 - 2^d e^{-(u-j)/2^d})^{k^d/d!} \tag{2.3.1}$$



(a) $k = 256$ (b) $t = 0$

Figure 2.3.3: Lower bounds on d for *k-XOR-LFSR*.

For $d \leq \log(u-j) - \log(\log(u-j)\log k + \log\log k)$, this lower bound is about $\geq 1/e$. Since $u \approx k$ and after neglecting the term $\log\log k$, this shows an approximate lower bound on $d(\{Trig\})$, i.e.,

$$\geq \log(k - t(\{Trig\})) - \log(\log(k - t(\{Trig\}))\log k)$$

This characterizes the stealthiness of the *k-XOR-LFSR*. In other words, the stealthiness of *k-XOR-LFSR* can be increased with k within the acceptable area overhead limits. Figure 2.3.3 shows the lower bounds on d for this HT for fixed values of k and t; if $t(\{Trig\})) = 0$, then $d(\{Trig\}) \geq \log k - 2\log\log k$. The HT can be designed for any point in the region above the lower bound. For the HT shown in Figure 2.3.2, clearly $\alpha(Trig) = 0$ since it *always* produces incorrect output immediately once the HT gets triggered.

2.4 Chapter Review

A first rigorous framework of Hardware Trojans is presented within which "Deterministic Trojans", the class H_D is introduced. In this chapter, several stealthiness parameters of the Hardware Trojans from H_D have been discovered which show that the current publicly known Hardware Trojans are the simplest ones in terms of stealthiness, and hence they represent just the tip of the iceberg at the huge landscape of Hardware Trojans. This framework leads us to design much more stealthy Hardware Trojans, and allows the Hardware Trojan research community to rigorously reason about the stealthiness of different Hardware Trojans and the effectiveness of existing countermeasures. The Hardware Trojan design principles introduced encourage and assist in designing new and even stronger countermeasures for highly stealthy and sophisticated Trojans.

Chapter 3

Advancing the State-of-the-Art in Hardware Trojans Detection

System on Chip (SoC) designers frequently use third party IP cores as black boxes instead of building these logic blocks from scratch, in order to save the valuable time and other resources. However, these third party IP cores can contain Hardware Trojans (HTs) which could potentially harm the normal functionality of the SoC (i.e. denial of service attack) or cause privacy leakage [13, 14, 15, 16]. These Trojans must be detected in pre-silicon phase, otherwise an adversary can infect millions of ICs through a Trojan affected IP core.

The IP core vendors generally do not offer the RTL level source code of the IP cores because of the proprietary reasons. Therefore, digital IP cores are typically offered in the form of a netlist of common digital logic gates and memory elements. Such IP cores are termed as 'closed source' IP cores. Even if the RTL source code of the IP core is available, it might not be feasible (in case of large IP cores) to manually inspect the source code for possible HTs. Hence, closed source IP cores impose additional challenges in identifying potentially malicious logic gate(s) from a netlist of hundreds of thousands of gates, without any knowledge of the RTL source code. This is analogous to finding a needle in a haystack.

This work has been published at *IEEE Transactions on Dependable and Secure Computing, 2017* [37].

Although Hardware Trojans research community has done tremendous efforts to develop effective countermeasures to detect several kinds of HTs, however still, the HT design research has always been one step ahead of the HT detection research. State of the art HT detection schemes namely Unused Circuit Identification (UCI) [21], VeriTrust [22], and FANCI [23] have shown that they can detect all HTs from the TrustHub [24] benchmark suite. On the other hand, researchers have also shown that these schemes can still be circumvented by carefully designing new HTs [25],[26]. A recent work, DeTrust [27], in fact introduces a systematic methodology of designing Trojans that can evade both VeriTrust and FANCI.

We notice that the fundamental reason behind the possibility of circumventing these HT detection techniques is the lack of rigorous analysis about the coverage of these techniques at their design time. Typically, HT detection techniques offer guaranteed coverage for a small *constant* set of *publicly known* HT benchmarks such as TrustHub. However, later on, one may design a HT which is slightly different than previously known HTs such that it can bypass the detection schemes. Therefore, clearly the HT detection techniques must target and provide security guarantees against certain behavioral traits resulting in a larger *class* of HTs instead of targeting specific known HTs. We limit our discussions in this work about digital IP cores and trigger activated digital HTs which have digital payloads. Also, since we want to detect any HTs inside the SoC before it gets fabricated, we restrict ourselves to the tools and algorithms which can detect HTs in pre-silicon phase. Hardware Trojans that are always active and/or exploit side channels for their payloads [38] [39] [40] [41] are out of scope of this work.

In chapter 2, we discovered several advanced properties (i.e. d, t, α, l) of trigger based deterministic HTs resulting in a large group called H_D [34]. These properties characterize the stealthiness of the HTs, and hence introduce the design principles that an adversary might follow to design sophisticated HTs in order to bypass the current HT countermeasures.

In this chapter, I present a powerful HT detection algorithm called ***Ha**rdware **T**rojan **C**atc**h**er* (HaTCh) that offers complete coverage and transparent security guarantees for the above mentioned large and complex class H_D of deterministic HTs. Figure 3.0.1 compares the coverages offered by different HT detection schemes for the HT class H_D. Although VeriTrust and FANCI can detect publicly known TrustHub HTs, it is unclear what security guarantees they offer outside TrustHub. HaTCh, on the other hand, is capable of

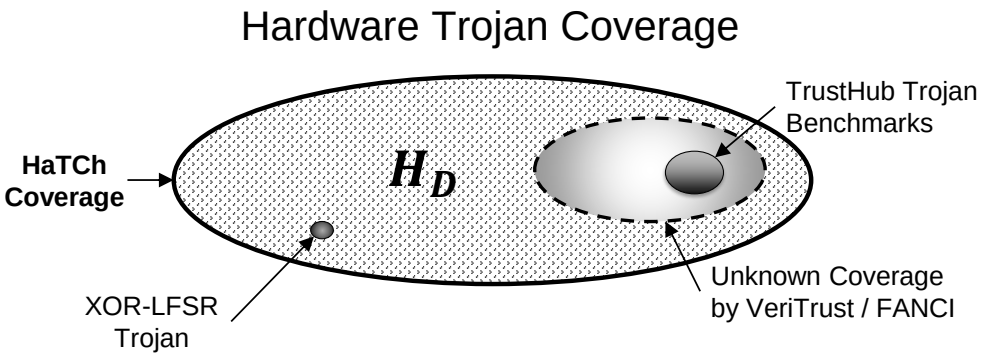


Figure 3.0.1: Class of Deterministic Hardware Trojans H_D, and detection coverages of existing Hardware Trojan countermeasures.

detecting any HT from H_D, whether the HT is publicly known or unknown. It is proved that HaTCh offers a negligible false negative rate $\leq 2^{-\lambda}$ and a controllable false positive rate $\leq \rho$ for user defined parameters λ and ρ.

Our proposed HT detection tool HaTCh works in two phases; a *learning phase* and a *tagging phase*. The learning phase performs logic testing on the IP core and produces a blacklist of all the unactivated (and potentially untrusted) transitions of internal wires. The tagging phase adds additional logic to the IP core to track these untrusted transitions. If these untrusted transitions – that are potentially related to a HT trigger circuit – ever occur in the lifetime of the circuit, an exception is raised that shows the activation of a HT. We have implemented HaTCh and our experimental as well as theoretical results demonstrate that:

- HaTCh detects all H_D Trojans in $poly(n)$ time where n is number of circuit wires and the degree of the polynomial corresponds to the stealthiness of the HT embedded in the circuit. In particular, all H_D Trojans from TrustHub are detected in linear time.
- HaTCh incurs low area overhead (on average 4.18% for a set of tested HT benchmarks).
- In cases where state of the art HT detection tools have a worst case exponential complexity in the number of circuit inputs, HaTCh offers orders of magnitude lower complexity.

The rest of this chapter is organized as follows. Section 3.1 provides necessary background of HTs,

briefly explains the HT class H_D w.r.t. its advanced properties followed by our threat model. Sections 3.2 and 3.3 present the basic algorithm and the detailed methodology of HaTCh, respectively. We rigorously analyze the security guarantees of HaTCh in section 3.4 and prove bounds on false negatives and false positives rates. In section 3.5 we compare HaTCh with state of the art HT detection schemes and section 3.6 shows the experimental evaluation.

3.1 Background & Threat Model

We first provide the necessary background and define various terminologies upon which we build our further discussions in this chapter. In particular, we briefly review the characteristics of the HT class H_D presented in chapter 2 which play a crucial role in the development of our detection algorithm called HaTCh.

3.1.1 Basic Terminologies

Some important terminologies from chapter 2 are as follows:

Definition 3.1.1. A **Hardware Trojan** (HT) can be defined as malicious *redundant* circuitry embedded inside a larger circuit, which results in data leakage or harm to the normal functionality of the circuit once activated.

A *trigger activated* HT activates upon some special event, whereas an *always active* HT remains active all the time to deliver the intended payload. Once activated, a HT can deliver its payload either through standard I/O channels and/or through the side channels.

Definition 3.1.2. Trigger condition is an event, manifested in the form of a particular boolean value of certain internal/external wires of the circuit, which activates the HT trigger circuitry.

Definition 3.1.3. Trigger signal or Trigger State is a collection of physical wire(s) which the HT trigger circuitry asserts in order to activate the payload circuitry once a trigger condition occurs.

Definition 3.1.4. Explicit malicious behavior refers to a behavior of a HT where the HT generated output is *distinguishable* from a normal output.

Definition 3.1.5. Implicit malicious behavior refers to a behavior of a HT where the HT generated output is *indistinguishable* from a normal output.

Definition 3.1.6. A set $\mathcal{T}$ of trigger states *represents* a HT if the HT always passes through one of the states in $\mathcal{T}$ in order to express implicit of explicit malicious behavior.

Definition 3.1.7. False Negative is a scenario when a HT detection tool identifies a circuit containing a HT as a Trojan-free circuit or transforms a circuit containing a HT into a circuit which still allows the HT to express implicit or explicit malicious behavior.

Definition 3.1.8. False Positive is a scenario when a HT detection tool identifies a Trojan-free circuit to be a Hardware Trojan-infected circuit.

3.1.2 Properties of H_D Hardware Trojans

A Hardware Trojan detection scheme without a well defined scope on the landscape of HTs fails to provide concrete security guarantees. For this reason, we first define the scope of our detection algorithm by restricting it to the class of trigger based deterministic Hardware Trojans, namely the class H_D introduced in chapter 2. H_D represents the HTs which are embedded in a digital IP core whose output is a function of only its input, and the algorithmic specification of the IP core can exactly predict the IP core behavior. Four crucial properties (d, t, α, l) of this class are also introduced which determine the stealthiness of a H_D HT having a set of trigger states $\mathcal{T}$. A brief highlight of these properties is as follows.

Trigger Signal Dimension $d(\mathcal{T})$ represents the number of wires used by HT trigger circuitry to activate the payload circuitry in order to exhibit malicious behavior. A large d shows a complicated trigger signal, hence it is harder to detect.

Payload Propagation Delay $t(\mathcal{T})$ is the number of cycles required to propagate malicious behavior to the output port *after* the HT is triggered. A large t means it takes a long time after triggering until the malicious behavior is seen, hence less likely to be detected during testing.

Implicit Behavior Factor $\alpha(\mathcal{T})$ represents the probability that given a HT gets triggered, it will not (explicitly) manifest malicious behavior; this behavior is termed as implicit malicious behavior. Higher

probability of implicit malicious behavior means higher stealthiness during testing phase.

Trigger Signal Locality $l(\mathcal{T})$ shows the spread of trigger signal wires of the HT across the IP core. Small l shows that these wires are in the close vicinity of each other. Large l means that these wires are spread out in the circuit and therefore it is harder to figure out exactly which wires form the trigger signal, hence the HT becomes harder to detect.

A Hardware Trojan can be represented by multiple sets of trigger states $\mathcal{T}$, each having their own d, t, α, and l values. The collection of corresponding quadruples (d,t,α,l) is defined as the achievable region of the Hardware Trojan. This allows us to define $H_{d,t,\alpha,l}$ as follows:

Definition 3.1.9. $H_{d,t,\alpha,l}$ is defined as all H_D type Trojans which can be represented by a set of trigger states $\mathcal{T}$ with parameters $d(\mathcal{T}) \leq d$, $t(\mathcal{T}) \leq t$, $\alpha(\mathcal{T}) \leq \alpha$ and $l(\mathcal{T}) \leq l$.

In the remainder of this chapter we develop the HT detection tool HaTCh which takes parameters d, t, α and l as input in order to detect HTs from $H_{d,t,\alpha,l}$.[1]

3.1.3 Threat Model

A third party IP core is provided in the form of a synthesized netlist which obfuscates the HDL source code.[2] The IP core vendor has embedded a trigger based HT in the IP core which delivers its payload via standard IO channels of the IP core. The HT could have been inserted directly by maliciously modifying the source code or by using some malicious tools to synthesize the RTL. This IP core is used in a larger design whose millions of chips are fabricated. The adversary wants to exploit this scalability to infect millions of fabricated chips by supplying an infected IP core. To prevent this, we test the IP core for potential HTs in *pre-silicon* phase, i.e. before integrating it into a larger design and fabricating it. Logic or functional testing, used by Design for Test (DFT) community for testing basic manufacturing defects, is one of the simplest methods to test the IP core for basic HTs which can be easily implemented using the existing simulation/testing tools. For the above reasons we restrict ourselves to analyzing logic testing based tools used in pre-silicon phase

[1] $H_{d,t,\alpha,l} \subseteq H_{d,t,\alpha}$ where $H_{d,t,\alpha}$ represents the group of HTs which are not constrained by a certain value of the locality parameter l.

[2] Notice that IP cores offered as HDL source can also benefit from HaTCh once they are synthesized.

for HT detection.

3.2 HaTCh Algorithm

In this section, we present the basic HT detection algorithm of HaTCh which uses *whitelisting* approach to discriminate the trustworthy circuitry of an IP core from its potentially malicious parts.[3] In order to detect any HT in a $Core \in H_{d,t,\alpha,l} \subseteq H_D$, HaTCh takes the following parameters as input:

Core:	The IP core under test.
$\mathcal{U}$:	A user distribution with a ppt sampling algorithm SAMPLE where each $User \leftarrow$SAMPLE$(\mathcal{U})$ is a pt-algorithm.
d,t,α,l:	Parameters characterizing the Hardware Trojan.
λ:	False negative rate $\leq 2^{-\lambda}$.
ρ:	Maximum acceptable false positive rate.

Algorithm 1 shows the operation of HaTCh which processes *Core* in two phases; a *Learning phase* and a *Tagging phase*.

3.2.1 Learning Phase

The learning phase performs k iterations of functional testing on *Core* using input test patterns generated by k users from a user distribution $\mathcal{U}$ and learns k independent blacklists $B_1, B_2, \ldots, B_k$ of unused wire combinations. Here k depends upon the desired security level and is a function of α and λ. If *Core* is found manifesting any explicit malicious behavior during the learning phase then the learning phase is immediately terminated. This produces an error condition and as a result, HaTCh does not execute its tagging phase and simply returns "Trojan-Detected" which indicates that the IP core contains a Hardware Trojan, and

[3]The English word *Hatch* means an opening of restricted size allowing for passage from one area to another. Our HaTCh framework provides this functionality by allowing certain parts of the circuit to operate normally while restricting the operation of others.

is rejected straightaway in the pre-silicon phase. On the other hand, if no explicit malicious behavior is observed during the learning phase, a union of all individual blacklists B_i produces a final blacklist B. Having a union of multiple independent blacklists minimizes the probability of incorrectly whitelisting (due to the implicit malicious behavior) a trigger wire(s) since the trigger wire(s) need to be whitelisted in all k learning phases in order for the Trojan to remain undetected.

3.2.2 Tagging Phase

Once the final blacklist B is available, the tagging phase starts. It transforms $Core$ to $Core_{Protected}$ by adding extra logic for each entry in the blacklist such that whenever any of these wire combinations is activated, a special flag will be raised to indicate the activation of a potential Hardware Trojan. In particular, a new output signal called $TrojanDetected$ is added to the $Core$. A tree of logic gates is added to $Core$ such that a logic 1 is propagated to $TrojanDetected$ output whenever any wire combination from B takes a 'blacklisted' value. The area overhead of tagging circuitry is $O(|B|d)$ where d represents the parameter passed to HATCH. Notice that the added logic can be pipelined to keep it off the critical path and hence it would not affect the design timing. The pipeline registers may delay the detection of the HT by $O(\log_2(|B|d))$ cycles, however for average sized IP cores, HaTCh can produce a significantly small B with reasonable run time. Consequently, the detection delay because of pipeline registers is only a few cycles.

We notice that logic testing based approach for Trojan detection is generally pretty straightforward, and has already been proposed in [36]. However, the main edge of HaTCh over existing schemes is that it can detect a much larger class H_D of HTs which, to say the least, cannot be efficiently detected by the existing schemes.

3.2.3 Example: Interaction of HaTCh Parameters

Figure 3.2.1 shows a simple HT embedded in a half adder circuit. The HT-free circuit in (shaded in green in Figure 3.2.1a) generates a sum $S = A \oplus B$ and a carry $C = A \cdot B$. The HT (highlighted in red) triggers when $A = B$ and produces incorrect result after one clock cycle, i.e. $S = B$ instead of $S = A \oplus B$.

The table in Figure 3.2.1b shows the typical parameters setting for this HT. Since the trigger signal T

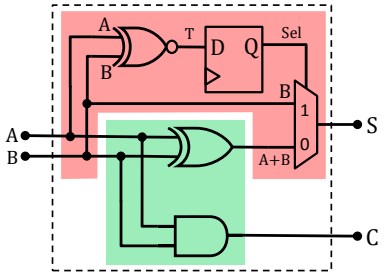
Algorithm 1 HaTCh Algorithm

procedure HATCH($Core, \mathcal{U}, d, t, \alpha, l, \lambda, \rho$)
  $k = \lceil \frac{\lambda}{log_2(1/\alpha)} \rceil$, $B = \emptyset$
  for all $1 \leq i \leq k$ **do**
    $B_i \leftarrow$ LEARN($Core, \mathcal{U}, d, t, l, \rho$)
    if $B_i =$ "Trojan-Detected" **then**
      return "Trojan-Detected"
    else
      $B = B \cup B_i$
    end if
  end for
  $Core_{Protected} =$ TAG($Core, B$)
  return $Core_{Protected}$
end procedure

only becomes 1 when the trigger condition is satisfied, the dimension d of this HT is $d = 1$. The locality parameter l only makes sense for HTs with $d > 1$, therefore currently it is not applicable. Once the HT is triggered, if the input during the next cycle is $(A, B) = (0, X)$ then the so called 'malicious output' $S = B$ is indistinguishable from the otherwise normal output of the circuit, hence resulting in implicit malicious behavior (cf. definition 3.1.5). The probability of such behavior given the HT has been triggered is 0.5, hence $\alpha = 0.5$. The malicious behavior of the HT can only happen 1 clock cycle after the trigger condition has occurred, therefore here $t = 1$.

Figure 3.2.1b also shows the effects of varying HaTCh parameters on the number of wires in the black-list. Here, we assume that the learning phase tests all possible input patterns (including those which exhibit implicit malicious behavior), except the ones which show explicit malicious behavior. The blacklist size also determines the false positives rate as well as the area overhead.

For $d = 1$, all the wires (including the trigger signal) are whitelisted because of implicit malicious behavior, hence $Blklst = 0$. This means the HT cannot be detected yet. In order to compensate for the high value of α, if d is set to 2 then HaTCh monitors all 2-wire combinations and captures pairs like $(A, Sel) = (1, 1)$ which detect HT activation. However this increases the blacklist size. For $d = 2$ and $l = \infty$, we see $Blklst = 10$ since *all* possible 2-wire combinations are monitored; whereas for $l = 1$, $Blklst = 4$ since the locality parameter forces HaTCh to only monitor the combinations of any two wires which are only one



(a) Half Adder with HT

	Typ.	*Change*	*Blklst*
d	1	$1 \rightarrow 2$	$0 \rightarrow 10$
l	N/A	$\infty \rightarrow 1$	$10 \rightarrow 4$
α	0.5	N/A	N/A

(b) Changes in Blacklist Size.

Figure 3.2.1: A simple HT: Trigger condition $A = B$; Normal output $S = A \oplus B$; Malicious output (1 cycle after trigger condition) $S = B$.

logic level apart from each other. Notice that the parameter t, in most cases, imposes overheads in terms of memory/time complexities but has minimal effect on the blacklist size for a long enough learning phase. Also, since a HT can be represented by multiple trigger 'states', e.g., depending upon the wires considered as 'trigger signal', hence one sees different values of t for different trigger states. E.g., for trigger states $(A,B) = (1,1)$, $T = 1$, and $Sel = 1$, we see $t = 1$, $t = 1$, and $t = 0$ respectively.

3.3 Detailed Methodology of HaTCh

In order to formally model and define Hardware Trojans, and to present a detailed implementation of HaTCh, we first provide a relaxed model for the input and output behavior of the IP cores.

3.3.1 IP Core

An IP core '*Core*' given as a gate-level netlist represents a circuit module $M = M^{Core}$ (with feed-back loops, internal registers with dynamically evolving content, etc.) that receives inputs (over a set of input wires) and produces outputs (over a set of output wires). We define the *state* of M at a specific moment in time (measured in cycles) as the vector of binary values on each wire inside M together with the values stored in each register/flip-flop. Here, the definition of state goes beyond just the values stored in the registers inside M: M itself may not even have registers that store state, M's state is a snapshot in time of M's combinatorial logic (which evolves over time). By S_i we denote M's state at clock cycle i.

User-Core Interaction

We model a user as a polynomial time (pt) algorithm[4] $User$ which, based on previously generated inputs and received outputs, constructs new input that is received by the IP core in the form of a new value. We assume (malicious) users who, due to (network) latencies, cannot observe detailed timing information (a remote adversary can covertly leak privacy over the timing channel if detailed timing information can be observed, which is out of scope of this model). In our model, we only consider trojans that can only deliver a malicious payload over the standard I/O channels in order to violate the functional specification of the core. This implies that only the message contents and the order in which messages are exchanged between the core and user are of importance.

We model this by restricting $User$ to a pt algorithm with two alternating modes; an *input generating mode* and a *listening mode*. During the jth input generating mode, some input message X_j is generated which is translated to a sequence $(x_k, x_{k+1}, \ldots, x_n)$ of input vectors for each clock cycle to the circuit module M which defines the IP core. During the jth listening mode of say Δ_j clock cycles, $User$ collects an output message Y_j that efficiently represents the sequence of output vectors $(y_g, y_{g+1}, \ldots, y_k, y_{k+1}, \ldots, y_n, \ldots, y_{n+\Delta_j})$ as generated by M during clock cycles from the end of the last input generating mode at clock cycle g onwards, i.e., the output generated during clock cycles $g, g+1, \ldots, n+\Delta_j$ (here, we write $x_i = \varepsilon$ or $y_i = \varepsilon$ if no input vector is given, i.e. input wires are undriven, or no output vector is produced). In other words, $User$ simply produces an input message X_j, waits to receive an output message Y_j, produces a new input message X_{j+1} and waits for the new output message Y_{j+1} etc. The X_j are produced as semantic units of input that arrive over several clock cycles at the IP core. Y_j concatenates all the meaningful ($\neq \varepsilon$) output vectors that were generated by the IP core since the transmission of X_j. This means that the view of the user is simply an ordered sequence of values devoid of any fine grained clock cycle information.

[4] Any random coin flips necessary are stored as a common reference string in the algorithm itself. Note that user is a deterministic algorithm.

3.3.2 Functional Specifications

We assume that the IP core has an algorithmic functional specification consisting of two algorithms: *CoreSim* and *OutSpec*. *CoreSim* is an algorithm that simulates the IP core at the coarse grain level of semantic output and input units:

- *CoreSim* starts in an initial state S'_0
- $(Y'_j, S'_j, \Delta_j) \longleftarrow CoreSim(X_j, S'_{j-1})$

CoreSim should be such that it does not reveal any information about how the IP core implements its functionality. It protects the intellectual property (implementation and algorithmic tricks etc.) of the IP core and only provides a specification of its functional behavior. States S'_j are not related to the states S_i that are snapshots of the circuit module M as represented by *Core*. States S'_j represent the working memory of the algorithm *CoreSim*. Notice that *CoreSim* also outputs Δ_j, the listening time needed to receive Y_j if a user would interact with M^{Core} instead of *CoreSim*.

The output specification *OutSpec* specifies which standard output channels should be used and how they should be used. Standard output channels are defined as those which can be configured by the hardware itself (by programming reserved registers etc.). E.g., a hardware trojan doubling the Baud rate (by overwriting the register that defines the UART channel) or a hardware trojan which unexpectedly uses the LED channel (by overwriting the register that programs LEDs), as implemented in [42], would violate *OutSpec*. Notice that side channel attacks are defined as attacks which use non-standard output channels and these attacks are not covered by *OutSpec*.

Emulation of M^{Core}

We assume that the *Core*'s gate-level netlist allows the user of the IP core to emulate its fine grained behavior (the state transition and output vector for each clock cycle), i.e., we assume an algorithm $Emulate$:

- $Emulate[Core]$ starts in an initial state S_0.
- $(y_i, S_i) \longleftarrow Emulate[Core](x_i, S_{i-1})$.

Algorithm 2 *User* interacts with $Emulate[Core]$ and verifies functional correctness and outputs the list of all the emulated states of M^{Core}.

procedure SIMULATE($Core, User$)
  $g, Y_0, j, States = 1, \varepsilon, 1, [\,]$
  $S_0, S'_0 = ResetStateCore, ResetStateSim$
  while $(X_j, U_j) \leftarrow User(Y_{j-1}, U_{j-1})$ **do**
    $(Y'_j, S'_j, \Delta_j) \leftarrow CoreSim(X_j, S'_{j-1})$
    $(x_k, \ldots, x_{k+n}) \leftarrow$ SEND(X_j)
    $(x_g, \ldots, x_{k-1}) = (\varepsilon, \ldots, \varepsilon)$
    $(x_{k+n+1}, \ldots, x_{k+n+\Delta_j}) = (\varepsilon, \ldots, \varepsilon)$
    for $i \leftarrow g, k+n+\Delta_j$ **do** ▷ Emulate
      $(y_i, S_i) \leftarrow Emulate[Core](x_i, S_{i-1})$
      if $y_i \neq \varepsilon$ **then** $Y_j = Y_j || y_i$
      end if
      $Append(States, S_i)$ ▷ Update *States*
    end for
    $j, g = j+1,\ k+n+\Delta_j+1$
    if $Y'_j \neq Y_j$ **then** ▷ Verification
      return ("Trojan-Detected", $\cdot$)
    end if
  end while
  return ("OK", $States$) ▷ All emulated states
end procedure

$Emulate[Core]$ behaves exactly as the circuit module M corresponding to *Core*, i.e. $Emulate[Core]$ and M are functionally the same. The main difference is that $Emulate[Core]$ parses the language in which *Core* is written: In practice, one can think of $Emulate[Core]$ as any post-synthesis simulation tool, such as Mentor Graphic's ModelSim [43] simulator, which can be used to simulate the provided IP core netlist *Core*. Notice the following properties of such a simulator tool; firstly it does not leak any information about the IP other than described by *Core* itself and secondly, it is inefficient in terms of (completion time) performance since it performs software based simulation, however it provides fine grained information about the internal state of the IP core at every clock cycle.

Simulation of User-Core Interaction

The user of the IP core is in a unique position to use $Emulate[Core]$ and verify whether its I/O behavior (over standard I/O channels) matches the specification $(CoreSim, OutSpec)$. The verification can be done automatically without human interaction: This will lead to the proposed HaTCh tool which uses (during a *learning phase*) $Emulate[Core]$ to simulate the actual IP core M^{Core} and verifies whether the sequence $(X_1, Y_1, X_2, Y_2, \ldots)$ of input/output messages to/from $User$ matches the output sequence $(Y'_1, Y'_2, \ldots)$ of *CoreSim* on input $(X_1, X_2, \ldots)$. Algorithm 2 shows a detailed description of this process (U_i indicates the current state or working memory of $User$).

Notice that $User$ in algorithm 2 can be considered as a meta user which runs several test patterns from different individual users one after another to test M^{Core}. This implies that SIMULATE is generic and can be applied to both a non-pipelined as well as a pipelined M^{Core}.

Functional Spec Violation

We consider H_D trojans, therefore, *CoreSim* is a non-probabilistic algorithm. This means that the output sequence $(Y'_1, Y'_2, \ldots)$ of *CoreSim* is uniquely defined (and next definitions make sense): We define the input sequence $X_1, X_2, \ldots, X_N$ to not violate the functional spec if it verifies properly in algorithm 2, i.e., if the emulated output (by $Emulate[Core]$) correctly corresponds to the simulated output (by *CoreSim*). If it does not verify properly, then we say that the input sequence $X_1, X_2, \ldots, X_N$ violates the functional spec.

3.3.3 Legitimate States & Projections

We first present a technical definition of t-legitimate states and d-dimensional projections which will help in explaining the process of whitelisting in the learning phase:

Definition 3.3.1. t-Legitimate States: Let $(w, States) \leftarrow \text{SIMULATE}(Core, User)$. Assuming *Core* is fixed, we define $W(User) = w$ and the set of t-legitimate states of $User$ as:

$$L_t(User) = \{States[1], \ldots, States[|States| - t]\}$$

(Since SIMULATE is deterministic, $L_t(User)$ and $W(User)$ are well-defined.)

Definition 3.3.2. Projections: We define a vector $\mathbf{z}$ projected to index set P as $\mathbf{z}|P = (\mathbf{z}_{i_1}, \mathbf{z}_{i_2}, \ldots, \mathbf{z}_{i_d})$ where $P = \{i_1, i_2, \ldots, i_d\}$ and $i_1 < i_2 < \cdots < i_d$. We call d the dimension of projection P and we define $\mathcal{P}_d$ to be the set of all projections of dimension d. We define a "set Z projected to $\mathcal{P}_d$" as

$$Z|\mathcal{P}_d = \{(P, \mathbf{z}|P) : \mathbf{z} \in Z, P \in \mathcal{P}_d\}.$$

Formally, a trigger state is a labelled binary vector, i.e., it is a pair $(P, \mathbf{x})$ where P denotes a projection and $\mathbf{x}$ is a binary vector; if *Core* is in state $\mathbf{z}$ and $\mathbf{z}|P = \mathbf{x}$ then the trojan gets triggered. Now let $\mathcal{T}$ be a set of trigger states/signals which *represents* the hardware trojan, i.e., M^{Core} manifests malicious behavior if and only if it has passed through a state in $\mathcal{T}$. Let $\mathcal{T}$ have dimension d and payload propagation delay t, i.e., the trojan always manifests malicious behavior within t clock cycles after "it gets triggered" by a trigger signal in $\mathcal{T}$ (cf. section 3.1.2). Then we know that a state in $L_t(User)|\mathcal{P}_d$ can only correspond to a trigger signal in $\mathcal{T}$ if the trigger signal produced implicit malicious behavior, i.e., $W(User) =$ "OK". In the next subsection, we use this notion of t-legitimate states to learn a blacklist of suspicious wires in the IP core.

3.3.4 Learning Phase Algorithm

Algorithm 3 describes the operation of a single iteration in HaTCh learning phase[5] (lines 3-10 in Algorithm 1). First a $User$ is sampled from $\mathcal{U}$ and at least $1/\rho$ test patterns generated by $User$ are tested on *Core*. All those internal states (wires) which are reached by *Core* during these tests are whitelisted and the rest of the states (wires) are considered to be the part of blacklist. This process is repeated until the blacklist size does not reduce any further, i.e. until $1/\rho$ consecutive user interactions (and their resulting states) do not reduce the blacklist anymore. This means that neither a false nor a true positive would have been generated if this blacklist was used for the tagging phase. For this reason ρ becomes, statistically, the estimated upper bound on the false positive rate. A detailed proof of this bound is presented in section 3.4.2.

[5] In our complexity analysis we assume white listing happens as soon as possible so that double work in lines 14-16 is avoided.

Algorithm 3 Learning Scheme

1: **procedure** LEARN($Core, \mathcal{U}, d, t, l, \rho$)
2: **if** I/O register does not match $OutSpec$ **then**
3: **return** "Trojan-Detected"
4: **else**
5: $B = \mathcal{P}_d \times \{0,1\}^d$, $User \leftarrow$ SAMPLE($\mathcal{U}$)
6: **repeat**
7: $B_{old} = B$
8: Steps from Algorithm 2 from line 2-3
9: **for** $m = 1$ **to** $1/\rho$ **do**
10: $(X_j, U_j) \leftarrow User(Y_{j-1}, U_{j-1})$
11: Steps from Algorithm 2 from line 5-18
12: **end for** ▷ If not aborted, this yields $States$
13: **for all** $P \in \mathcal{P}_d$ **do** ▷ Perform Whitelisting
14: **for all** $1 \leq i \leq |States| - t$ **do**
15: $B = B \setminus \{(P, States[i]|P)\}$
16: **end for**
17: **end for**
18: **until** $|B| \neq |B_{old}|$ ▷ Until no change in B
19: **return** B ▷ The Blacklist
20: **end if**
21: **end procedure**

The blacklist B generated by algorithm 3 is equal to

$$(\mathcal{P}_d \times \{0,1\}^d) \setminus (L_t(User)|\mathcal{P}_d) \tag{3.3.1}$$

for the sampled $User$ (line 5 in algorithm 3).

Notice that B may contain two types of wires; first the wires specifically related to the Hardware Trojan circuitry, and second some redundant wires which did not excite during the learning phase either because of insufficient user interactions or because of logical constraints of the design. The second type of wires in the blacklist would lead to false positives. However, the controlled nature of the HaTCh algorithm allows the user to control the false positives rate by trading off with the learning phase run time.

3.3.5 Security Guarantees of HaTCh

If HaTCh does not detect a functional spec violation during its learning phase, then the blacklist produced by HaTCh is the union of k independent blacklists corresponding to k independent users $User$ with $W(User) =$ "OK", see (3.3.1). If the set of trigger states $\mathcal{T}$ is not a subset of this union, then each of the k blacklists must exclude at least one trigger signal from $\mathcal{T}$ and therefore $(L_t(User)|\mathcal{P}_d) \cap \mathcal{T} \neq \emptyset$ for each of the corresponding k users $User$. The probability that both $W(User) =$ "OK" as well as $(L_t(User)|\mathcal{P}_d) \cap \mathcal{T} \neq \emptyset$ is at most α (by Bayes' rule) for a $H_{t,\alpha,d}$ trojan. We conclude that the probability that the set of trigger states $\mathcal{T}$ is not a subset of the blacklist produced by HaTCh is at most $\alpha^k \leq 2^{-\lambda}$. So, the probability that the tagging circuitry will detect all triggers from $\mathcal{T}$ is at least $1-2^{-\lambda}$. A detailed proof of this bound is presented in section 3.4.1.

3.3.6 Computational Complexity of HaTCh

The computational complexity of HaTCh depends upon λ, α, d, ρ, and $n = |Core|$. HaTCh performs k iterations in total during the learning phase in algorithm 1, where $k = \lceil \lambda / \log_2(1/\alpha) \rceil$. The length of each iteration is determined by the number $|\mathcal{P}_d \times \{0,1\}^d| = \binom{n}{d} 2^d \leq (2n)^d$ of possible triggers of dimension d, and the desired false positive rate ρ: in the worst case every $1/\rho$ user interactions in an iteration may only reduce the blacklist by one possible trigger, hence, the length of each iteration is $O((2n)^d/\rho)$. Whereas, in each iteration, the search space to find and whitelist the projections from is $|\mathcal{P}_d| = \binom{n}{d} \leq n^d$. Therefore the overall computational complexity of HaTCh is given by:

$$O\left(\frac{\lambda}{\log_2(1/\alpha)} \cdot \frac{(2n^2)^d}{\rho}\right) \tag{3.3.2}$$

We conclude that HaTCh runs in $poly(n)$ time where the degree of the polynomial corresponds to the maximum value of the trigger signal dimension d of the HT that can be detected in this much time. Notice that in general, the degree of this polynomial is much smaller than the total number of wires that define the HT itself.

In order to reduce the computational complexity, we exploit the *locality* in gate level circuits (cf. section 3.1.2). Let n_l denote the maximum number of wires in the locality of any of the n wires of the circuit for

parameter l, then the projections search space drastically reduces to $n \cdot \binom{n_l}{d-1} \leq n \cdot n_l^{d-1}$ since $n_l \ll n$ Hence, the overall complexity from (3.3.2) is reduced to:

$$O\left(\frac{\lambda}{\log_2(1/\alpha)} \cdot \frac{n^2(2n_l^2)^{d-1}}{\rho}\right)$$

A user might not know the exact parameters of the HT to be detected, however he can find a reasonable configuration through some heuristic. One such heuristic is outlined below. Notice that the adversary doesn't need to know the parameters selection strategy. Hence HaTCh forces specific efforts form the adversary making his job harder.

Let the user has a computational budget that can be invested in the HaTCh learning phase for desired values of λ and ρ. Select t as the IP core latency (i.e. pipeline delay) given in the IP core specification. Set $\alpha = 0.5$ since this potentially gives the most advantage to the adversary. l is proportional to the HT footprint which is usually limited, therefore a small value of l should be picked. Solving the complexity equation with all the above mentioned parameters for d give the value of d.

3.4 Rigorous Security Analysis of HaTCh

For medium to large sized practical circuits, it becomes almost impossible to precisely calculate the probability of implicit malicious behavior, i.e. α (cf. section 3.1.2). However, α can be experimentally analyzed and approximated by *experimentally observed* implicit behavior factor α'. We define α' as follows:

$Core \in H_{d,t,\alpha'}^{\text{SAMPLE}}$ if and only if it is represented by a set of trigger states $\mathcal{T}$ with $t(\mathcal{T}) \leq t$ and $d(\mathcal{T}) \leq d$ such that

$\mathcal{C}1$) There exists a $User$ and a state S in the set of all reachable states of M^{Core} interacting with $User$ such that $S \in \mathcal{T}$. I.e., $Core$ is indeed capable of manifesting malicious behavior.

$\mathcal{C}2$) For all $User$, $\text{SIMULATE}(Core, User)$ outputs $W(User)$ such that:

$$Prob\Big(W(User) = \text{“OK”} \Big| \big(L_t(User)|\mathcal{P}_d\big) \cap \mathcal{T} \neq \emptyset\Big) \leq \alpha'$$

where the probability is over $User \leftarrow \text{SAMPLE}(\mathcal{U})$.

Depending upon the sampling algorithm SAMPLE, $\alpha' \geq \alpha$ (bad for detection) or $\alpha' \leq \alpha$ (good for detection). However, we assume that SAMPLE algorithm samples close to the actual distribution: If the distributions $\text{SAMPLE}(\mathcal{U})$ and $\mathcal{U}$ are equal and $t = 0$ in $\mathcal{C}2$, then α' becomes the actual implicit behavior factor α.

Notice that reducing t to 0 in $\mathcal{C}2$ allows a trigger signal to manifest also during the last t states produced by $\text{SIMULATE}(Core, User)$. For such a trigger signal the payload will less likely manifest in malicious behavior as it will have less than t cycles to progress to the output, hence, for such trigger signals $W(User) =$ "OK" is more likely. This means that the overall probability in $\mathcal{C}2$ (calculated as a weighted average over trigger signals which manifest during the last t cycles and trigger signals which manifest before the last t cycles) increases as t reduces to 0. So, if distribution $\text{SAMPLE}(\mathcal{U})$ equals $\mathcal{U}$, then $H_{d,t,\alpha'} \subseteq H^{\text{SAMPLE}}_{d,t,\alpha'}$.

The next section proves an upper bound of false negative probability for $H^{\text{SAMPLE}}_{d,t,\alpha'}$. So, if SAMPLE algorithm samples close to the actual distribution then this result holds for $H_{d,t,\alpha'}$.

3.4.1 False Negatives

As discussed in the previous section, if SAMPLE algorithm samples close to the actual distribution, then the theorem below also holds for $H_{d,t,\alpha}$ with α interpreted as the implicit behavior factor:

Theorem 3.4.1. *For an IP core in $H^{\text{SAMPLE}}_{d,t,\alpha}$ analyzed by* HATCH*$(Core, \mathcal{U}, d, t, \alpha, l, \lambda, \rho)$, the probability of a false negative is upper bounded by α^k, where α is the experimentally observed implicit behavior factor of the IP core and k is the number of iterations in HaTCh learning phase.*

Proof. The final blacklist B is a union of the k blacklists $B_1, B_2, \ldots, B_k$ learned by HaTCh, i.e. $B = B_1 \cup B_2 \cup \cdots \cup B_k$. Each blacklist B_i initially includes all d-dimensional projections of wires, i.e. $B_i = \mathcal{P}_d \times \{0,1\}^d$. The projections that activate are then whitelisted from B_i one by one during the functional testing.

Let a HT be represented by a set of trigger states $\mathcal{T}$. A false negative under HaTCh is only possible if:

1. The IP core passes the functional testing during the learning phase, and

2. The tagging circuitry cannot flag a malicious behavior, i.e. $\exists_{T \in \mathcal{T}}\ T \notin B$

Since the blacklist $B = B_1 \cup B_2 \cup \cdots \cup B_k$, therefore for a false negative the trigger state $T \in \mathcal{T}$ must be whitelisted from all sub-blacklists B_i for $1 \leq i \leq k$. For $U_i \leftarrow \text{SAMPLE}(\mathcal{U})$ and $(W(U_i), States_i) \leftarrow \text{SIMULATE}(Core, U_i)$, this probability is given by:

$$Prob(FN) = Prob\left(\underset{1 \leq i \leq k}{\forall} W(U_i) = \text{“OK”} \wedge \underset{1 \leq i \leq k}{\forall} T \notin B_i\right)$$

By definition of t-Legitimate states, the probability that $\exists_{T \in \mathcal{T}}\ T \notin B_i$ is the same as the probability that $L_t(U_i) \cap \mathcal{T} \neq \emptyset$. Therefore, $Prob(FN)$ is:

$$\begin{aligned} &= \prod_{1 \leq i \leq k} Prob\Big(W(U_i) = \text{“OK”} \wedge L_t(U_i) \cap \mathcal{T} \neq \emptyset\Big) \\ &\leq \prod_{1 \leq i \leq k} Prob\Big(W(U_i) = \text{“OK”} \;\Big|\; L_t(U_i) \cap \mathcal{T} \neq \emptyset\Big) \\ &\leq \alpha^k \end{aligned}$$

by definition of experimentally observed implicit behavior factor α. □

3.4.2 False Positives

Consider a user distribution $\mathcal{U}$ and a user $U \leftarrow \mathcal{U}$ which interacts with the IP core *Core*, i.e.,

$$(w, States) \leftarrow \text{SIMULATE}(Core, U).$$

In the following discussion, a subscript 's' of *Prob* represents the scenario when *Prob* is over users U sampled by SAMPLE algorithm, i.e., $U \leftarrow \text{SAMPLE}(\mathcal{U})$. No subscript '$s$' represents the scenario when *Prob* is over users U drawn from $\mathcal{U}$.

In order to do a precise analysis of false positives, we make the following assumptions:

Significance Assumption: Algorithm LEARN outputs a blacklist $B = B_1 \cup B_2 \cup \cdots \cup B_k$ where (1) the com-

plement of B_j equals the set

$$\bar{B}_j = \bigcup_{1 \le i \le L_j} States[i] \tag{3.4.1}$$

for some index L_j and (2) B_j is not decreased for another $p\Delta$ cycles after L_j (parameters p and Δ will be chosen when we describe our statistical assumptions), i.e.

$$\forall_{1 < i \le p\Delta} States[L_j + i] \notin B_j. \tag{3.4.2}$$

By $(3.4.2)_j$ and $(3.4.1)_j$ we denote properties (3.4.2) and (3.4.1) for B_j.

We assume that there is significance to the event that each iteration of LEARN produces a L_j and B_j for which it is likely that for a period of another $p\Delta$ states, no decrease in the B_j is observed. I.e., if one would look for another multiple of $p\Delta$ states, we assume that again with significant probability no decrease in B_j will be observed. In mathematical notation we assume that

$$Prob_s\left(\forall_{1 \le j \le k} (3.4.2)_j \Big| \forall_{1 \le j \le k} (3.4.1)_j\right) \ge 1 - \beta \tag{3.4.3}$$

where $1 - \beta$ represents the significance.

Statistical Assumptions:

(a) We assume that Δ is sufficiently large such that, for all i, the following relation holds:

$$\begin{aligned} &Prob\Big(States[i] = S_1 \wedge States[i+\Delta] = S_2\Big) \\ &= Prob\Big(States[i] = S_1\Big) \cdot Prob\Big(States[i+\Delta] = S_2\Big) \end{aligned}$$

In other words, Δ is large enough such that if the IP core is interacting with U then any two states in the resulting sequence of state transitions of the IP core that are Δ transitions away from each other are uncorrelated.

For practical circuits, this is a valid assumption. E.g. for a pipelined AES circuit, if the pipeline takes less than Δ cycles, then the internal states of the IP core Δ cycles apart correspond to different independent plain texts input to the AES circuit and therefore the internal states that are Δ cycles apart correspond to

independent identical distributed values on the internal wires/registers (hence, independent states).

(b) We assume that sampling of the user U by SAMPLE(U) algorithm is done close to the real distribution $\mathcal{U}$, i.e., for all i, $Prob_s(States[i] = S) \approx Prob(States[i] = S)$.

(c) Finally, we assume that $Prob(States[i] = S)$ is independent of i. Continuing the AES example above, this is a valid assumption; the states resulting from independent identical distributed plain texts being input consecutively to an AES circuit, each have the same individual distribution (even though they may be correlated if they are less than Δ transitions away from each other). For completeness, we notice that if we only assume periodic behavior ($Prob(States[i_0] = S) = Prob(States[i_1] = S)$ if i_0 equals i_1 modulo a period), then the results below generalize.

Given this assumption, we can define

$$\rho(B) = Prob(States[i] \in B)$$

for sets B.

Analysis: Let $(3.4.4)_j$ denote the property

$$\underset{1<i\leq p}{\forall} States[L_j + i\Delta] \notin B_j. \tag{3.4.4}$$

By the significance assumption,

$$1 - \beta \leq Prob_s \left(\underset{1\leq j\leq k}{\forall} (3.4.2)_j \middle| \underset{1\leq j\leq k}{\forall} (3.4.1)_j \right)$$

which, since $(3.4.2)_j$ implies $(3.4.4)_j$, is at most

$$Prob_s \left(\underset{1\leq j\leq k}{\forall} (3.4.4)_j \middle| \underset{1\leq j\leq k}{\forall} (3.4.1)_j \right). \tag{3.4.5}$$

Notice that all U_j are sampled independently, therefore (3.4.5) is equal to the product

$$\prod_{j=1}^{k} Prob_s((3.4.4)_j|(3.4.1)_j).$$

We derive

$$\begin{aligned}
&Prob_s((3.4.4)_j|(3.4.1)_j)\\
&= Prob_s\left(\underset{1<i\le p}{\forall} States[L_j+i\Delta] \notin B_j\right) \text{ (by (a))}\\
&= \prod_i Prob_s\Big(States[L_j+i\Delta] \notin B_j\Big) \text{ (by (a))}\\
&\approx \prod_i Prob\Big(States[L_j+i\Delta] \notin B_j\Big) \text{ (by (b))}\\
&= \prod_i \Big(1-\rho(B_j)\Big) \text{ (by assumption (c))}\\
&= \big(1-\rho(B_j)\big)^p
\end{aligned}$$

The above derivations prove

$$1-\beta \le \Big(\prod_j (1-\rho(B_j))\Big)^p.$$

Since a product of real numbers in [0,1] is at most the product of their averages,

$$1-\beta \le \left(\Big(\sum_j \big(1-\rho(B_j)\big)/k\Big)^k\right)^p = \Big(1-\sum_j \rho(B_j)/k\Big)^{kp}.$$

After reordering terms we obtain

$$\sum_j \rho(B_j) \le k\big(1-(1-\beta)^{1/(kp)}\big) \tag{3.4.6}$$

By using $1-(1-x)^y \le y(x+(1/(1-x)-1)^2/2)$ for $0 \le x \le 1$ and $0 \le y \le 1$, the right hand side of (3.4.6) is at most

$$k(\beta+(\beta/(1-\beta))^2/2)/(kp) = (\beta+(\beta/(1-\beta))^2/2)/p$$

and we conclude

$$\sum_j \rho(B_j) \le (\beta+(\beta/(1-\beta))^2/2)/p. \tag{3.4.7}$$

A state leads to a false positive if it is in the blacklist B (which means that it is being flagged) while it is not one of the trigger states of a set of trigger states $\mathcal{T}$ representing a potential hardware Trojan. By using (3.4.7) we get

$$\begin{aligned}
&Prob(State \text{ leads to a FP}) \\
&\leq Prob(State \in B \setminus \mathcal{T}) \\
&\leq Prob(State \in B) \\
&= Prob_s(State \in B) \text{ (by assumption (b))} \\
&\leq \textstyle\sum_j Prob_s(State \in B_j) \text{ (by the union bound)} \\
&= \textstyle\sum_j \rho(B_j) \\
&\leq (\beta + (\beta/(1-\beta))^2/2)/p
\end{aligned}$$

By choosing $p = 1/\rho$ and significance $1-\beta = 1/2$ we get:

Theorem 3.4.2. *For statistical assumptions (a), (b), and (c), and significance* $1/2$, HATCH*(Core,* $\mathcal{U}, d, t, \alpha, l, \lambda, \rho$*) leads to a tagging circuitry which flags non Hardware Trojan trigger events (i.e., false positives) with probability at most* ρ.

We notice that the analysis leading to the above theorem states that the more likely our observations in HaTCh, i.e., significance $1-\beta$ is closer to 1, then the more we believe a tighter upper bound $\beta(1+(\beta/(1-\beta)^2)/2)\rho \approx \beta\rho$ on the probability of false positives.

3.5 Comparison with Existing Techniques

3.5.1 Detection Capability Comparison

We determine the detection capability of a countermeasure for a particular class of HTs (in this case H_D) by two factors: (1) the false negatives rate, and (2) the false positives rate. Clearly, a countermeasure which offers zero false negatives rate by treating each and every circuit as "malicious" and therefore resulting in $\approx 100\%$ false positives rate is of no use, and similarly vice versa. In terms of detection capability, the best

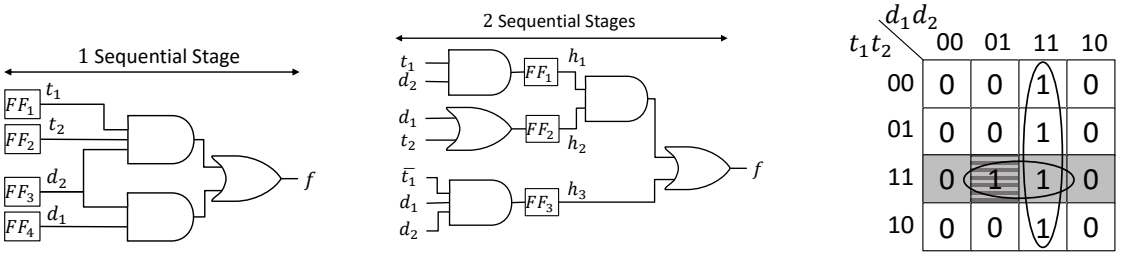
countermeasure is the one which offers minimum of the two rates.

UCI: *Unused Circuit Identification* (UCI) [21] tries to distinguish minimally used logic in the design from more frequently used parts of the circuit. The intuition here is that a HT almost always remains inactive in the circuit to pass the functional verification. However, due to functional verification constraints, the whole designs cannot be activated and analyzed in optimal time, and hence the scheme identifies large portions of the design to be 'unused' and consider them as potential HTs. This results in a high false positive rate, and recent works have even succeeded in defeating this scheme [25], [26]. Later techniques have improved over UCI both in terms of security and performance.

VeriTrust: *Veritrust* [22] proposed by Zhang *et al.* detects HTs by identifying the inputs in the combinational logic cone that seem redundant for the normal functionality of the output wire under non-trigger condition. In order to detect the redundant inputs, it first performs functional testing and records the activation history of the inputs in the form of sums-of-products (SOP) and product-of-sums (POS). Then it further analyzes these unactivated SOPs and POSs to find the redundant inputs. However, because of the functional verification constraints, VeriTrust can see several unactivated SOPs and POSs and thus regard the circuit to be potentially infected resulting in false positives.

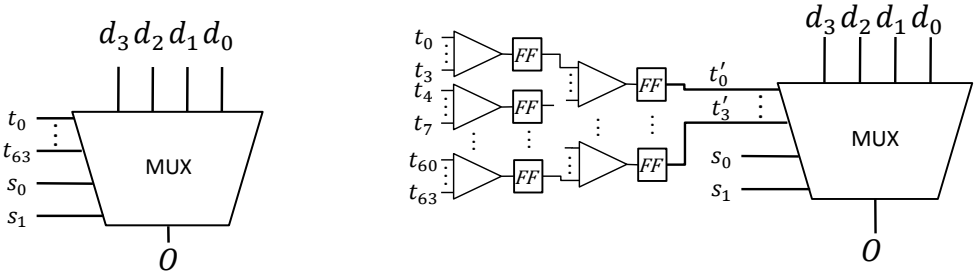
FANCI: Waksman *et al.* presents FANCI [23] which applies boolean function analysis to flag suspicious wires in a design which have weak input-to-output dependency. A *control value* (CV), which represents the percentage impact of changing an input on the output, is computed for each input in the combinational logic cone of an output wire. If the mean of all the CVs is lower than a threshold, then the resulting output wire is considered malicious This is a probabilistic method where the threshold is computed with some heuristic to achieve a balance between false negatives and the false positives rate. A very low threshold may result in a high false positive rate by considering most of the wires (even non-malicious ones) as malicious, whereas a high threshold may actually result in false negatives by considering a HT related (malicious) wire to be 'not' malicious.

DeTrust: One of the most recent works *DeTrust* [27] presents a systematic way to design new HTs which cannot be detected by either FANCI or VeriTrust. Notice that, at any time, both VeriTrust and FANCI



(a) HT detectable by VeriTrust (b) HT undetectable by VeriTrust (c) Simplified form of f

Figure 3.5.1: DeTrust defeating VeriTrust: Normal function $f = d_1d_2$, Trojan affected function $f = d_1d_2 + t_1t_2d_2$, Trigger condition $(t_1, t_2) = (1, 1)$.



(a) HT detectable by FANCI (b) HT undetectable by FANCI

Figure 3.5.2: DeTrust defeating FANCI: A 4-to-1 MUX is transformed into a malicious MUX with 64 additional inputs where trigger condition is one out of 2^{64} possible input patterns.

only monitor the combinational logic between two registers (i.e. one sequential stage). In other words, to decide whether an input to a flipflop (FF) is malicious or not, VeriTrust and FANCI consider the combinational logic cone (i.e. circuitry starting from the outputs of the previous FF stage) driving this wire to be a standalone circuit and then run their respective checks on it. DeTrust exploits this limitation and designs new HTs whose circuitries are intermixed with the normal design and distributed over multiple sequential stages such that FANCI/VeriTrust, while observing a single sequential stage, would consider them non-malicious.

Figure 3.5.1 and Figure 3.5.2 show example HT designs that DeTrust presents to defeat VeriTrust and FANCI respectively. In Figure 3.5.1, DeTrust splits the combinational logic cone of the original (less stealthy) HT design (Figure 3.5.1a) into two sequential stages by inserting flipflops (Figure 3.5.1b). Now VeriTrust considers f to be only a function of h_1, h_2 and h_3; and as none of these inputs are redundant to define f, VeriTrust identifies f to be a non-malicious output. Similarly in Figure 3.5.2, the 64 trigger inputs

of the original HT design (Figure 3.5.2a) are spread over multiple sequential stages (Figure 3.5.2b) to defeat FANCI. Hence, even though FANCI/VeriTrust can detect all TrustHub HT benchmarks, DeTrust shows how an adversary can design new HTs to defeat the existing countermeasures.

Extended VeriTrust & FANCI: In order to detect its newly designed HTs, DeTrust proposes extensions of VeriTrust and FANCI. In this work, we refer to these extensions as **VeriTrustX** and **FANCIX** respectively. The key idea is to monitor the circuits up to multiple sequential stages at a time, while ignoring any FFs in between. Notice that in the worst case, one may need to monitor all the sequential stages of the design starting from the external inputs. However, if one has some knowledge about the number of sequential stages that a HT can be spread over, VeriTrustX or FANCIX only need to monitor that many number of stages at a time. For example, VeriTrustX and FANCIX will consider the HTs from Figure 3.5.1b and Figure 3.5.2b respectively to be one big combinational logic block. Therefore the added stealthiness by the FFs is eliminated and hence both the techniques will be able to detect the respective HTs.

Formal Methods: Rajendran *et al.* proposed to use Bounded Model Checking (BMC) in order to formally verify that the untrusted IP cores do not contain any HTs [44]. First the IP core vendor and the user agree upon certain properties that should be satisfied by the design (e.g., "No data corruption of critical registers"), then the user performs BMC to verify these properties. A violation of any property shows existence of a HT in the IP core.

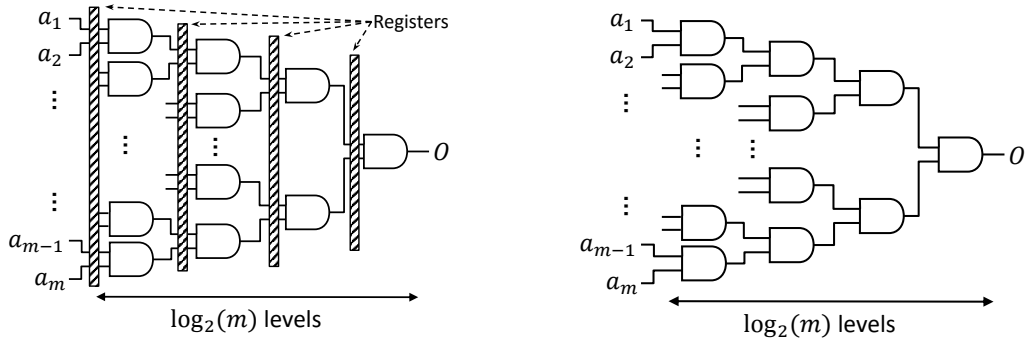
The very nature of formal verification poses some limitations on the detection capability of this technique. (1) Although it is an efficient scheme to detect "known" or "expected" HTs as one can write properties to verify that such HTs do not exist, however it cannot detect HTs which are not envisioned or anticipated by the user. In other words, one needs to know *all* possible malicious behaviors (e.g., not only the various ways data corruption but also data leakage etc.) that can possibly be exhibited by the HTs so that properties can be developed to perform BMC and detect such HTs. This itself is a hard task to accomplish which is not required in HaTCh. (2) The absence of runtime checking has the advantage of having no false positives, but more critically it could potentially allow false negatives since the BMC would only search for HTs for a relatively small number of clock cycles as compared to the activation interval of (say) a large counter based

time-bomb type HT. On the other hand, HaTCh prioritizes the elimination of any false negatives by performing runtime checking which, as a side effect, might introduce false positives. Given two equivalently long verification/learning phases, the formal methods technique would reduce the probability of false negatives while maintaining no false positives, whereas HaTCh would reduce the probability of false positives while maintaining no false negatives (because of runtime checking). Since we consider false negatives to be more critical than false positives, HaTCh offers a better solution to the HT detection problem.

TPAD: *Trojan Prevention and Detection architecture* (TPAD) is a runtime checking approach uses an output characteristic predictor and a checker function to verify the predicted out of a circuit against its actual output [45]. Although this approach shares some similar aspects with HaTCh (e.g., runtime monitoring, additional hardware checkers etc.), however its major limitation is that it assumes the RTL of the IP core is trusted and available to the user. HaTCh, on the other hand, addresses a stronger adversarial model where the RTL source code of the design is not provided to the user (for proprietary reasons) and can even be untrusted.

Trusted RTL: *Trusted RTL* [36] uses functional testing in combination with equivalence checking assisted by automatic test pattern generation (ATPG) tools to find HTs whose trigger signals behave like *"stuck at"* fault wires. Although this technique could potentially detect all simplistic HTs which have dimension $d = 1$, however, higher dimensional HTs ($d > 1$) cannot be simply detected by finding *"stuck at"* faults. The reason is that the trigger signal of higher dimensional HTs comprises of multiple wires, typically none of which is stuck at a single value. Hence Trusted RTL in its current form offers minimal coverage for H_D class of HTs as compared to HaTCh.

Ali *et al.* [46] present a HT to leak the secret key of AES encryption module through faulty cipher texts. The HT gets triggered upon a specific sequence of three plain texts, and selects a malicious clock source to insert a glitch in the clock signal to the AES module causing the faulty cipher text. Interestingly, in our framework, this HT belongs to the simplistic $d = 1$ type of HTs since the multiplexer select line that chooses the malicious clock source is a single wire which remains stuck at '0' until the HT activation. Clearly, such a trigger signal of a HT will be blacklisted by HaTCh.



(a) Original implementation. (b) Seen by VeriTrustX/FANCIX.

Figure 3.5.3: An m-input AND gate implemented by 2-input AND gates.

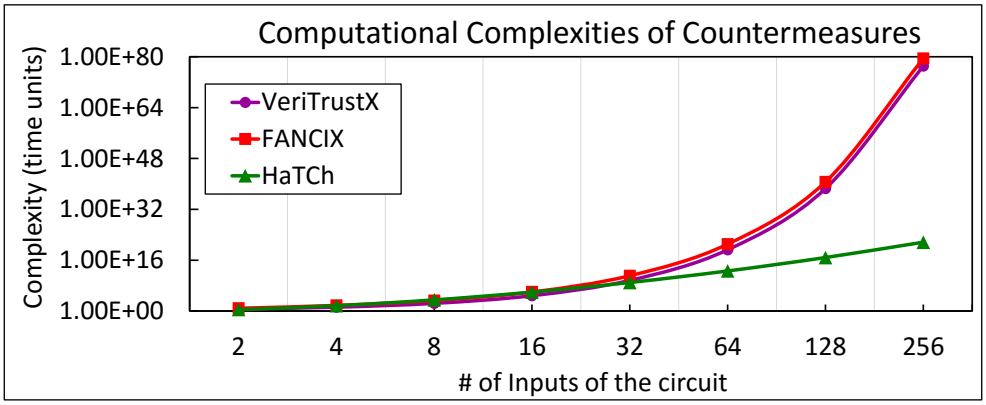


Figure 3.5.4: Computational complexity comparison of different techniques.

3.5.2 Computational Complexity Comparison

We now compare the computational complexities of existing state of the art HT detection schemes, namely VeriTrustX and FANCIX, with the complexity of HaTCh. Out of the several different countermeasures discussed in the last section, since only VeriTrustX and FANCIX are potentially the closest ones to HaTCh in terms of the HT coverage that they offer, therefore we only compare the complexities of these two techniques with HaTCh.

In order to analyze the worst case complexities of different HT detection techniques, we consider a simple circuit shown in Figure 3.5.3a as our *"test subject"*, i.e., as if this is the HT circuit that needs to be

detected[6]. This circuit implements a m-input pipelined AND gate using a tree of 2-input AND gates and registers, where the tree has $\log_2(m)$ levels (i.e. sequential stages). Since without any knowledge of the HT stealthiness, both VeriTrustX and FANCIX will need to monitor all the sequential stages of the design, therefore it will be considered a purely combinational logic block as shown in Figure 3.5.3b.

To avoid any false negatives caused by the implicit malicious behavior[7] (cf. section 3.1.1, definition 3.1.5), VeriTrustX must monitor the activation history of each entry in the truth table instead of the terms in SOP/POS form of the boolean function. This leads to an exponential computational complexity i.e. $\Omega(2^m)$ as the truth table has 2^m entries in total.

Similarly FANCIX also needs to go through each entry of the truth table to compute the control value (CV) of a single input, and this process is repeated for each of the m inputs. This leads to a computational complexity $\Omega(m2^m)$. We do notice that the basic version of FANCI suggests to reduce the complexity by randomly selecting a reasonable number of entries in the large truth table to compute the control values. However, this optimization can potentially lead to a higher probability of false negatives.

Since there are only $\log_2(m)$ sequential stages over which this type of HT can be spread, HaTCh can detect this HT by using $d = \log_2(m)$ in the worst case. Therefore, if n represents the total number of wires in the circuit then HaTCh needs to monitor a search space of n^d wire combinations (i.e. projections). Hence HaTCh can detect the HT with a complexity $O(n^{\log_2(m)})$ or $O(2^{d\log_2(n)})$ as compared to $\Omega(2^m) = O(2^{2^d})$ offered by existing techniques. In other words, the worst case complexity of HaTCh is orders of magnitude lower than that of VeriTrustX and FANCIX.

This exponential vs. double exponential behavior (in d) is depicted in Figure 3.5.4 which shows computational complexities of different countermeasures against the number of inputs $m = 2^d$ of the circuit from Figure 3.5.3. Notice that both horizontal and vertical axes in this plot show a logarithmic scale. For this particular circuit, n is given by $n = 2m - 1$, i.e. the sum of all internal nodes (wires) and the leafs of the tree. As it can be seen, HaTCh offers orders of magnitude lower complexity, particularly for circuits with

[6]Notice that, for real, this circuit is not meant to implement a HT functionality. The sole purpose of selecting this simple circuit is to conduct the worst case complexity analysis.

[7]For $(t_1, t_2, d_1, d_2) = (1, 1, 1, 1)$ tested on circuit from Figure 3.5.1, the HT gets triggered but since $f = 1$, the verifier doesn't detect the HT.

number of inputs $m > 32$. Therefore for practical circuits, such as encryption engines which have 128 or more inputs, HaTCh is more feasible as it scales better with the number of inputs (i.e. circuit size) whereas the complexities of both VeriTrustX and FANCIX shoot up rapidly.

Although in practice, the HT is a part of a larger circuitry, we notice that the complexities of HaTCh and the other techniques scale in the same fashion since the additional wires of the larger circuit increase the search space for all these detection techniques.

3.6 Evaluation

In this section, we evaluate our HaTCh tool for Trusthub [24] benchmarks, Trojan designs proposed by DeTrust which defeat VeriTrust and FANCI, and also for a newly designed k-XOR-LFSR Trojan. We first analyze the Trusthub benchmarks w.r.t. HaTCh framework. Then we briefly describe our experimental setup and methodology including some crucial optimizations implemented in HaTCh to minimize the area overhead. Finally we present and discuss the experimental results. Notice that section 3.5.1 and 3.5.2 explain from a rational reasoning perspective how HaTCh compares with various existing techniques including FANCIX and VeriTrustX if they are pushed to cover the similar HT ground (namely H_D) as HaTCh. It has been intuitively shown that these techniques suffer from high computational complexity. Furthermore, due to public unavailability of the source codes of these techniques (e.g. FANCI), we cannot provide an experimental side-by-side comparison with HaTCh.

3.6.1 Characterization of TrustHub Benchmarks

The definitional framework introduced in chapter 2 provides a concrete characterization of the TrustHub [24] Hardware Trojans benchmark suite. We revisit that analysis here briefly, which helps us to pick concrete parameter settings for HaTCh in order to detect these HTs. Since HaTCh is a tool to be used in pre-silicon phase, we are only interested in the class of deterministic HTs which use standard I/O channels to deliver their payloads, i.e. the class H_D as per the terminology used in chapter 2.

Table 3.6.1 presents this class with subclassifications based on the specific properties of the HTs, namely

Table 3.6.1: Classification of Trusthub Benchmarks w.r.t. the definitional framework of chapter 2.

Hardware Trojans Class H_D	d	t	α	***Benchmarks***
	1	0	$1/2^{32}$	BasicRSA-T{100, 300}
			0.5	s15850-T100, s38584-T{200, 300}
			0-0.25	wb_conmax-T{100, 200, 300}
			0-0.87	RS232-T{100, 800, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1900, 2000}
		1	0.5	b15-T{300,400}
			0.5-0.75	s35932-T{100, 200}
			0-0.06	RS232-T{400, 500, 600, 700, 900, 901}
		2	0.5	vga-lcd-T100, b15-T{100, 200}
			0.87	s38584-T100
		3	$1/2^{32}$	BasicRSA-T{200, 400}
			0.5	s38417-T100
		5	0.99	s38417-T200
		7	0.5	RS232-T300
		8	0.5	s35932-T300

d, t, and α. In our experiments, we set the input parameters to HaTCh according to the values listed in this table. All these Trojans happen to have the simplest trigger signal having a dimension $d = 1$, whereas the t and α values are observed estimated values. The methodology adapted to estimate t and α values is summarized as follows. To determine t values, simply the minimum number of registers between the trigger signal wires(s) and the output port of the IP core is computed. In order to estimate α values, first the smallest chain of logic gates starting from the trigger signal wire(s) till the output port of the IP core (ignoring any registers in the path) is found. Then for each individual logic gate, the probability of propagating a logic 1 (considering that the trigger wire(s) get a logic 1 upon a trigger event) is computed. Finally an aggregate probability of propagation is computed by multiplying all the probabilities of each logic gate in the chain, which gives the value $1-\alpha$. This estimated α relates to the *experimentally observed* implicit behavior factor α' in our security analysis (cf. section 3.4).

3.6.2 Experimental Setup & Methodology

We first test five different benchmarks from RS232 and seven from *s-Series* (i.e., s15850, s35932 and s38417) benchmark groups (all of which together form a diverse collection) using the parameters t and

α as listed in Table 3.6.1. Since these HTs have the dimension $d = 1$, we also set the parameter $d = 1$ for HaTCh. Also, we set the locality parameter l to infinity (unless specified otherwise), essentially forcing HaTCh to exhaustively monitor all possible d-wire combinations across the whole design and hence demonstrating the worst case area overheads. For all our experiments, we set the maximum acceptable false positive rate ρ to be 10^{-5}. HaTCh detects all tested benchmarks (i.e. zero false negatives), and the resulting area overheads of tagging circuitries are presented in the results section. Notice that s38417-T300 is a side channel based HT, but since it does not get triggered in the learning phase, HaTCh is still able to detect it.

Even though these benchmarks have a maximum dimension $d = 1$ which means that they can be detected already by using $d = 1$ in HaTCh, we test certain RS232 benchmarks with parameters $d = 2$ and locality $l = 1$ in order to estimate the area overhead for these parameter settings. These results are also presented later in this section.

HaTCh tool works on a synthesized gate level netlist of the IP core. We use Synopsys Design Compiler [47] to synthesize the RTL design. Next, we perform post-synthesis simulations with self checking testbenches using Mentor Graphic's ModelSim [43] simulator. The benchmark is given random test patterns as inputs (ATPG tools can also be used to generate patterns) and the self checking testbench verifies the correct behavior, and the simulation trace of each wire is dumped into a file upon successful verification. HaTCh parses the simulation dump file using an automated script to generate a blacklist. Initially all possible transitions of all the wires of the circuit are blacklisted. Then, every transition read by the script from the simulation file is removed from the blacklist which eventually leads to a final blacklist containing only the untrusted transitions of certain wires. Based on the final blacklist, additional logic is added to flag the blacklisted transitions.

HaTCh Optimizations to shrink the blacklist

The blacklist generated in the learning phase contains two different types of wires. First, the wires of HT trigger circuit and the actual trigger signal wires which constitute the true blacklist. Second, the wires which are in the blacklist because of the low coverage of the input test patterns. The blacklist might also contain some redundant wires from either or both the two categories. For example if the input(s) and the output of

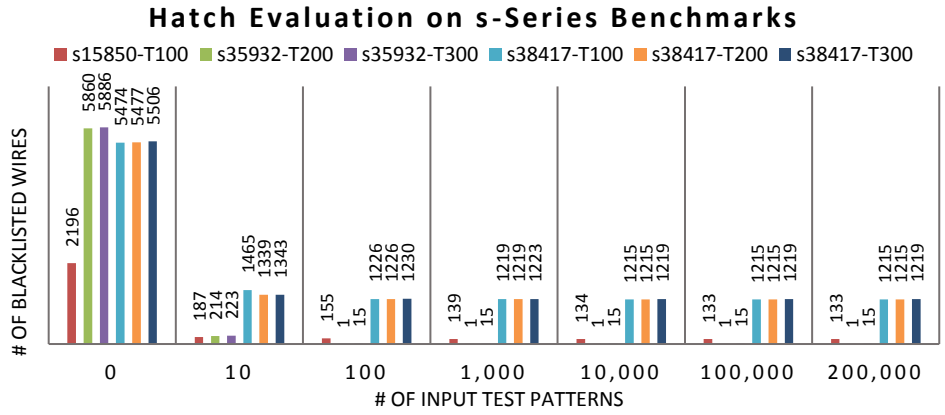


Figure 3.6.1: Blacklist size of s-Series with $d = 1$

certain logic elements (gates, buffers) exist in the blacklist at the same time, then it is sufficient to keep only the input in the blacklist provided that changing this input will affect the output, e.g. in case of logic buffers and inverters etc. Redundancy is also caused by certain wire combinations which are logically not possible, e.g. the combination of the input and output wires of an inverter is never going to have a value of 00 or 11, but HaTCh with $d = 2$ will still blacklist these combinations. Our HaTCh tool identifies and eliminates such redundancies from the blacklist wherever possible, e.g. for all buffers, logic gates and flipflops in the design, which in turn reduces the area overhead of the tagging circuit.

3.6.3 Experimental Results

TrustHub s-Series Benchmarks

Figure 3.6.1 shows the size of the blacklists sampled after different numbers of input patterns for s-Series benchmarks. For each benchmark, the blacklist size decreases rapidly with the number of input patterns until it reaches a state when most of the wires in the design are already whitelisted and no more wires are eliminated from the blacklist by further testing. E.g. the blacklists for the s35932 group become stable already after only 100 input patterns. Whereas s38417 group achieves the stable state after 10,000 input patterns. Only s15850 group takes longer to become stable.

Table 3.6.2: Area Overhead for s-Series with $d = 1$

Benchmark	***Size***	***Area Overhead***	
		Pipelined	***Non-Pipelined***
s15850-T100	2180	4.17%	2.11%
s35932-T200	5442	0.02%	0.02%
s35932-T300	5460	0.16%	0.09%
s38417-T100	5341	15.22%	7.62%
s38417-T200	5344	15.21%	7.62%
s38417-T300	5372	15.25%	7.63%
Average		8.34%	4.18%

Table 3.6.2 shows the area overhead incurred by HaTCh for s-Series benchmarks both for non-pipelined and pipelined tagging circuitries. The size of benchmarks (gates+registers) is shown under *Size*. On average, we see an overhead of 8.34% and 4.18% for pipelined and non-pipelined circuitries respectively. For some benchmarks, we see significantly high overhead than others which is most likely because of the fact that the random input test patterns do not provide enough coverage for some of the benchmarks. We observe that the optimizations performed by HaTCh (cf. section 3.6.2) reduce the overheads by ≈ 4.5 times as compared to the un-optimized tagging circuitries.

TrustHub RS232 Benchmarks

We first test five RS232 benchmarks (namely RS232-T$\{100, 300, 500, 600, 700\}$) which have dimension $d = 1$. The blacklists produced by HaTCh for these benchmarks only contains a single wire, i.e. the trigger signal. Hence, these benchmarks do not incur any additional area overhead.

We also test some RS232 benchmarks (i.e. RS232-T$\{300, 1200, 1300\}$) using parameters $d = 2$ and locality $l = 1$ just to get a real estimate of HaTCh overheads for higher dimensions, even though these benchmarks belong to $d = 1$ HT subclass. Table 3.6.3 shows the area overheads of these benchmarks. We see that even with $d = 2$, the overheads for these benchmarks are reasonably small. Since the computed blacklists for these benchmarks are very small, the tagging circuitry would only consist of only 2 to 3 logic levels. Therefore we do not need a pipelined tagging circuitry for these benchmarks.

Table 3.6.3: Area Overhead for RS232 with $d = 2$, $l = 1$

Benchmark	***Size***	***Area Overhead (non-pipelined)***
RS232-T300	280	2.50%
RS232-T1200	273	0.73%
RS232-T1300	267	0.75%

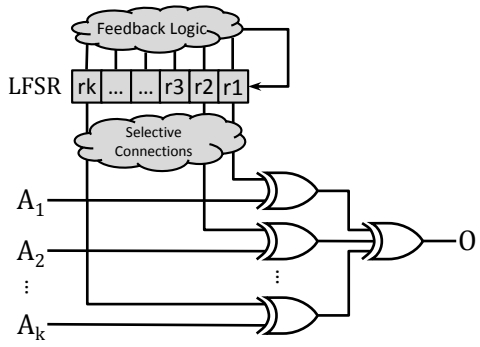


Figure 3.6.2: *k-XOR-LFSR* Hardware Trojan (cf. section 2.3.2).

New Trojans by DeTrust

As TrustHub benchmarks are quite simple and also outdated, therefore in order to demonstrate the strength of HaTCh tool, we implement the trojans proposed by DeTrust [27] (which defeat VeriTrust and FANCI) and test them with our HaTCh tool.

After synthesis on a standard ASIC library, the FANCI-defeating trojan from Figure 3.5.2b results in a trojan with trigger dimension $d = 1$. Hence, it is very straightforward to detect this trojan using the parameter $d = 1$ for HaTCh. In our experiment, after a learning phase of only $100,000$ random unique input patterns, the blacklist produced by HaTCh contains only one wire which is the trigger signal of this trojan.

The trojan defeating VeriTrust as depicted in 3.5.1b can be detected by HaTCh using the parameter $d = 2$ under the assumption that the implicit malicious behavior does not occur during the learning phase. Whereas without this assumption, HaTCh is still able to detect this trojan with $d = 3$ even if the implicit malicious behavior occurs.

k-XOR-LFSR Trojan

HaTCh can also detect the *k-XOR-LFSR* Trojan (cf. section 2.3.2) shown in Figure 3.6.2. As an example, we implemented the *k-XOR-LFSR* trojan for $k = 4$ which leads to a trigger dimension $d = 2$. Therefore, HaTCh requires the parameter $d = 2$ in order to detect it. The corresponding area overhead for this HT in our experiment is negligible as for now we tested it as a standalone circuit. However, depending upon the IP core in which this HT is embedded, the area overhead will vary in proportion to the IP core size. Notice that if frequency dividers are used to carefully slow down the clock frequency driving the *k-XOR-LFSR* HT, even a small k value can take several cycles before the HT is actually triggered.

3.7 Chapter Review

We provide a powerful Hardware Trojan detection algorithm called HaTCh which detects *any* Hardware Trojan from H_D, a large and complex class of "deterministic Hardware Trojans". HaTCh detects all H_D Trojans with a time complexity that is polynomial in number of circuit wires, where the degree of the polynomial corresponds to the stealthiness of the Hardware Trojan embedded in the circuit. In particular, all H_D Trojans from the TrustHub benchmark suite are detected in linear time.

We conclude that our work opens up new avenues for the Hardware Trojans research community to come up with other and possibly more efficient algorithms and methodologies to detect complex Trojans that are still publicly unknown.

Chapter 4

PrORAM: Dynamic Prefetcher for Oblivious RAM

As cloud computing becomes more and more popular, privacy of users' sensitive data is a huge concern in computation outsourcing. One solution to this problem is to use *tamper-resistant hardware* and secure processors. In this setting, a user sends his/her encrypted data to the trusted hardware, inside which the data is decrypted and computed upon. The final results are encrypted and sent back to the user. The trusted hardware is assumed to be tamper-resistant, namely, an adversary is not able to look inside the chip to learn any information. Many such hardware platforms have been proposed, including Intel's TPM+TXT [4], which is based on TPM [1, 2, 3], eXecute Only Memory (XOM) [5, 6, 7] and Aegis [8, 9].

While an adversary cannot access the internal states inside the tamper-resistant hardware, information can still be leaked through main memory accesses. Although all the data stored in the external memory can be encrypted to hide the data values, the memory access pattern (i.e., address sequence) may leak information. Existing attacks ([17]) show that the control flow of a program can be learned by observing the main memory access patterns. This may leak the sensitive private data.

Completely preventing leakage from the memory access pattern requires the use of Oblivious RAM

This work has been published at *Proceedings of the 42nd Annual International Symposium on Computer Architecture* [48].

(ORAM). ORAMs were first proposed by Goldreich and Ostrovsky [18], and there has been significant follow-up work that has resulted in more and more efficient cryptographically-secure ORAM schemes [49, 50, 51, 52, 53, 54, 55, 56, 57, 19, 58]. The key idea which makes ORAM secure is to translate a single ORAM read/write into accesses to multiple randomized locations. As a result, the locations touched in each ORAM read/write would have exactly the same distribution and be indistinguishable to an adversary.

The cost of ORAM security is performance. Each ORAM access needs to touch multiple physical locations which incurs one to two orders of magnitude more bandwidth and latency when compared to a normal DRAM. Path ORAM [19], the most efficient and practical ORAM system for secure processors so far, still incurs at least $30\times$ more latency than a normal DRAM for a single access. This results in $2-10\times$ performance slowdown [10, 20].

Traditionally, *data prefetching* [59, 60] has been used to hide long memory access latency. Data prefetching uses the memory access pattern from history to predict which data block will be accessed in the near future. The predicted block is prefetched from the memory before it is actually requested to hide the access latency.

Although it might seem that prefetching should be very effective with ORAM since ORAM has very high access latency, in reality prefetching does not work on ORAM when the program is memory bound. The main reason is that unlike DRAM, whose bottleneck is mainly memory latency, ORAM's bottleneck is in both latency and bandwidth. Prefetching only works when DRAM has extra bandwidth, therefore does not work well for ORAM (cf. Section 4.2.1).

In this work, we enable ORAM prefetching by exploiting locality inside the ORAM itself, which is very different from traditional prefetching techniques. At first glance, exploiting data locality and obfuscation seem contradictory: on one hand, obfuscation requires that all data blocks are mapped to random locations in the storage system. On the other hand, locality requires that certain groups of data blocks can be efficiently accessed together. One might argue that ORAM is inherently poor in terms of locality. We challenge this intuition in this work by exploiting data locality in ORAM without sacrificing provable security.

We propose a novel ORAM prefetcher called PrORAM (pronounced as *'Pro-RAM'*), which introduces a *dynamic super block* scheme. We demonstrate that it achieves the same level of security as normal Path

ORAM, and comprehensively explore its design space. Our dynamic super block scheme detects data locality in programs at *runtime*, and exploits the locality without leaking information on the access pattern.

In particular, the following contributions are made:

1. We study traditional data prefetching techniques in the context of ORAM, and observe that they do not work well for ORAM.
2. A dynamic super block scheme is proposed. The micro-architecture of the scheme is discussed in detail, and the design space is comprehensively explored.
3. Our simulation results show that PrORAM improves Path ORAM performance by 20.2% (upto 42.1%) over the baseline ORAM for memory bound Splash2 benchmarks, 5.5% for SPEC06 benchmarks, and 23.6% and 5% for YCSB and TPCC in DBMS benchmarks respectively. This is more than twice the performance gain offered by an existing static super block scheme.

The rest of the chapter is organized as follows: Section 4.1 provides the necessary background of ORAM in general and Path ORAM in particular. Section 4.2 presents ORAM prefetch techniques and discusses a previously proposed scheme called static super block. A dynamic super block scheme is introduced in Section 4.3. The design space is explored, security is shown and hardware complexity is analyzed in detail. Section 5.3 evaluates different optimizations proposed in the chapter and the related work is presented in Section 4.5.

4.1 Background

4.1.1 Oblivious RAM

ORAM ([18]) is a data storage primitive which hides the user's access pattern such that an adversary is not able to figure out what data the user is accessing by observing the address transferred from the user to the external untrusted storage (we assume DRAM in this work). A user accesses a sequence of program addresses $A = (a_1,\ a_2,\ \ldots,\ a_n)$, which will be translated to a sequence of ORAM accesses $S = (s_1,\ s_2,\ \ldots,\ s_m)$, where a_i is the program address of the i^{th} access and the value of a_i should be hidden from the adversary. s_i

is the physical address used to access the data storage engine. The value of s_i is exposed to the adversary. Given any two access sequences A_1 and A_2 of the same length, ORAM guarantees that the transformed access sequences S_1 and S_2 are computationally indistinguishable. In other words, the ORAM physical access pattern (S) is independent of the logical access pattern (A). Data stored in ORAMs should be encrypted using probabilistic encryption to conceal the data content and also hide which memory location, if any, is updated. With ORAM, an adversary should not be able to tell (a) whether a given ORAM access is a read or write, (b) which logical address in ORAM is accessed, or (c) what data is read from/written to that location.

In this work, we focus on Path ORAM [19], which is currently the most efficient ORAM scheme for limited client (processor) storage, and, further, is appealing due to its simplicity.

4.1.2 Path ORAM

Path ORAM [19] has two main hardware components: the *binary tree storage* and the *ORAM controller* (cf. Figure 4.1.1).

Binary tree stores the data content of the ORAM and is implemented on DRAM. Each node in the tree is defined as a *bucket* which holds up to Z data blocks. Buckets with less than Z blocks are filled with *dummy blocks*. To be secure, all blocks (real or dummy) are encrypted and cannot be distinguished. The root of the tree is referred to as level 0, and the leafs as level L. Each leaf node has a unique leaf label s. The path from the root to leaf s is defined as path s. The binary tree can be observed by any adversary and is in this sense not trusted.

ORAM controller is a piece of trusted hardware that controls the tree structure. Besides necessary logic circuits, the ORAM controller contains two main structures, a *position map* and a *stash*. The *position map* is a lookup table that associates the program address of a data block (a) with a path in the ORAM tree (path s). The *stash* is a piece of memory that stores up to a small number of data blocks at a time.

At any time, each data block in Path ORAM is mapped (randomly) to some path s via the position map. Path ORAM maintains the following invariant: *if data block a is currently mapped to path s, then a must be stored either on path s, or in the stash* (see Figure 4.1.1). Path ORAM follows the following steps when a request on block a is issued by the processor.

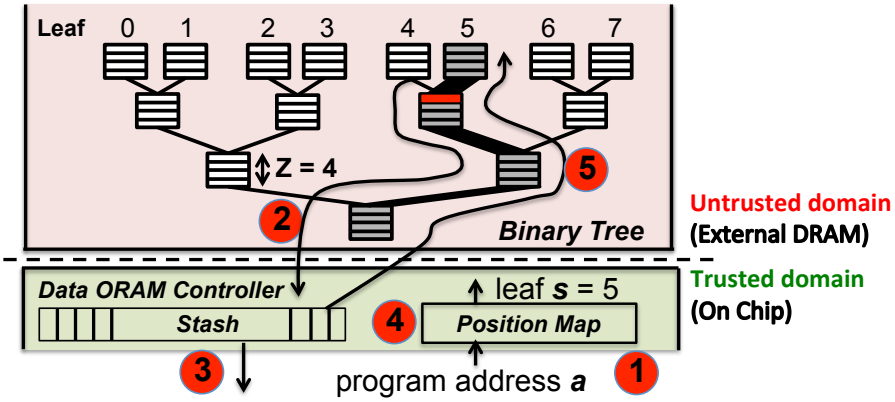


Figure 4.1.1: A Path ORAM for $L = 3$ levels. Path $s = 5$ is accessed.

1. Look up the position map with the block's program address a, yielding the corresponding leaf label s.
2. Read all the buckets on path s. Decrypt all blocks within the ORAM controller and add them to the stash if they are real (i.e., not dummy) blocks.
3. Return block a to the secure processor.
4. Assign a new random leaf s' to a (update the position map).
5. Encrypt and evict as many blocks as possible from the stash to path s. Fill any remaining space on path s with encrypted dummy blocks.

Step 4 is the key to Path ORAM's security. This guarantees that a random path will be accessed when block a is accessed later and this path is independent of any previously accessed random paths (*unlinkability*). As a result, each ORAM access is random and unlinkable regardless of the request pattern.

4.1.3 Recursive Path ORAM

In practice, the position map is usually too large to be stored in the trusted processor. Recursive ORAM has been proposed to solve this problem [56]. In a 2-level recursive Path ORAM, for instance, the original position map is stored in a second ORAM, and the second ORAM's position map is stored in the trusted processor (Figure 4.1.1(b)). The above trick can be repeated, i.e., adding more levels of ORAMs to further reduce the final position map size at the expense of increased latency. The recursive ORAM has a similar

organization as OS page tables.

Unified ORAM [61] is an improved and state-of-the-art recursion technique. It leverages the fact that each block in a position map ORAM stores the leaf labels for multiple data blocks that are consecutive in the address space. Therefore, Unified ORAM caches position map ORAM blocks to exploit locality (similar to the TLB exploiting locality in page tables). To hide whether a position map access hits or misses in the cache, Unified ORAM stores both data and position map blocks in the same binary tree. In this work, we use unified ORAM as our baseline ORAM design.

4.1.4 Background Eviction

In Steps 4 and 5 of the basic Path ORAM operation, the accessed data block is remapped from the old leaf s to a new random leaf s', making it likely to stay in the stash. In practice, this may cause blocks to accumulate in the stash and finally overflow. It has been proven that the stash overflow probability is negligible for $Z \geq 6$ [19]. For smaller Z, *background eviction* [20] has been proposed to prevent stash overflow.

The ORAM controller stops serving real requests and issues background evictions (*dummy accesses*) when the stash is full. A background eviction reads and writes a random path s_r in the binary tree, but does not remap any block. During the writing back phase (Step 5 in Section 4.1.2) of Path ORAM access, all blocks that are just read in can at least go back to their original places on s_r, so the stash occupancy cannot increase. In addition, the blocks that were originally in the stash are also likely to be written back to the tree (they may share a common bucket with s_r that is not full of blocks). Background eviction is proven secure in [20].

4.1.5 Timing Channel Protection

The original ORAM definition in [18] does not protect against timing attacks. Timing information includes when an ORAM access happens, the run time of the program, etc. For example, by observing that a burst of memory accesses happen, an adversary may be able to tell that a loop is being executed in the program. By counting the length of the burst, sensitive private information may be leaked.

In practice, periodic ORAM accesses are needed to protect the timing channel [10]. Following prior work, we use O_{int} as the public time interval between two consecutive ORAM accesses. ORAM timing behavior is completely determined by O_{int}. If there is no pending memory request when an ORAM access needs to happen due to periodicity, a dummy access will be issued (the same operation as background eviction). If one is willing to leak a few bits, timing channel protection schemes that allow for dynamically-changing O_{int} may be attractive [62], since they provide better performance. These schemes can be used with the techniques proposed in this work if small data leakage is allowed.

4.1.6 Path ORAM Limitation

Clearly, Path ORAM is far less efficient compared to insecure DRAM. Under typical settings for secure processors (gigabytes of memory and 64- to 128-byte blocks), Path ORAM has a 20-30 level binary tree (note that adding one level doubles the capacity). In practice, Z is usually 3 or 4 [57, 20]. This indicates that for each ORAM access, about 60-120 blocks need to be read **and** written, in contrast to a single read **or** write operation in an insecure storage system. Since a single ORAM access saturates the available DRAM bandwidth, it brings no benefits to serve multiple ORAM requests in parallel.

Recursive/unified ORAMs introduce additional overheads of accessing multiple levels of ORAMs. This overhead hurts both performance and energy efficiency. In total, Path ORAM incurs roughly two orders of magnitude more bandwidth and one order of magnitude more latency than DRAM. This leads to up to an order of magnitude slowdown in a secure processor [20]. Although no study has looked into the energy overhead of ORAM, we expect that the hundreds of blocks transferred to and from Path ORAM binary tree will result in proportionally larger energy consumption.

4.2 ORAM Prefetch: Super Block

As access latency is the main bottleneck in ORAM, a natural solution that comes to mind is to apply latency hiding techniques to ORAM. In this section, data prefetching is studied in the context of Path ORAM. We will show in Section 4.4.2 that traditional prefetching techniques do not work for ORAM and therefore new

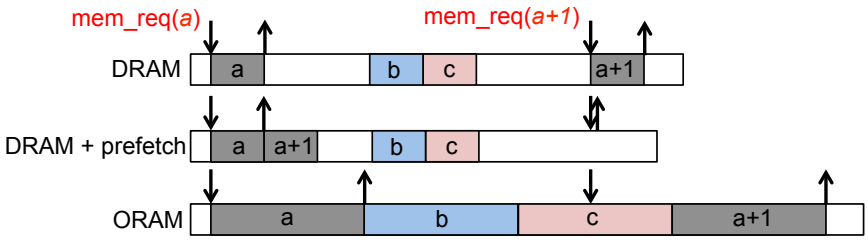


Figure 4.2.1: Data prefetching on DRAM and ORAM.

techniques are required.

4.2.1 Traditional Data Prefetch

Figure 4.2.1 shows the basic idea of traditional data prefetching. When block a is accessed, if the prefetcher predicts that block $a+1$ will also be accessed in the near future, then block $a+1$ is prefetched from the DRAM. When the real request to $a+1$ arrives, the data is already in the cache and the DRAM does not need to be accessed again. Prefetching moves memory accesses out of the critical path of execution thus leading to overall speedup of the program.

When DRAM is replaced with ORAM, however, the situation changes. The much longer latency (more than $30\times$) and lower throughput (2 orders of magnitude) of ORAM leads to two effects. First, it is not useful to overlap multiple ORAM accesses, since a single ORAM access already fully utilizes the entire DRAM bandwidth (cf. Section 4.1.6). Second, for memory bound applications, ORAM requests line up in the ORAM controller and there is no idle time for prefetching. In other words, prefetching is likely to block normal requests and hurt performance.

In this section, a new prefetch technique specifically designed for ORAM is proposed. The technique is called *Super Block*.

4.2.2 General Idea of Super Block

The notion of *super block*, first proposed in [20], tries to exploit spatial locality in ORAM. In particular, it tries to load more than one block from the path in a single ORAM access. The blocks that are loaded

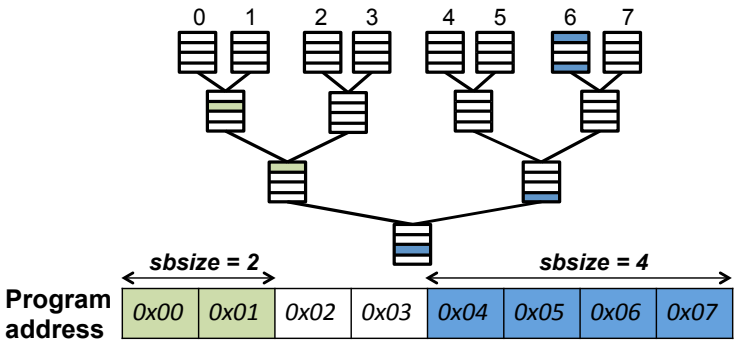


Figure 4.2.2: Super block construction. Blocks whose addresses are different only in the last k address bits can be merged into a super block of size $n = 2^k$. All blocks in a super block are mapped to the same path.

together are called a *super block*. According to Section 4.1.2, this requires that all the blocks belonging to a super block be mapped to the same path.

In this work, we only consider super blocks that consist of data blocks adjacent in program address space. Also, we only consider super blocks of size 2^k by merging blocks that differ only in the last k address bits. We define the size of a super block as the number of data blocks in it, denoted as *sbsize*. For example, in Figure 4.2.2, block 0x00 and block 0x01 can be merged into a super block of size 2; Blocks from 0x04 to 0x07 can be merged into a super block of size 4. However, block 0x03 and 0x04 cannot be merged because their addresses are not properly aligned.

Whenever one block in a super block is accessed, all the blocks in that super block are loaded from the path and remapped to a same random path. The block of interest is returned to the processor and the other blocks are *prefetched* and put into the LLC (Last Level Cache). The idea is that the prefetched blocks may be accessed in the near future due to spatial data locality, which saves some expensive Path ORAM accesses.

The super block scheme maintains the invariant that blocks in the same super block are mapped to the same path in the binary tree (Figure 4.2.2). This guarantees that all the blocks belonging to the same super block can be found during a single ORAM access (they may or may not reside in the same bucket). It is important to note that although a super block is always read out as a unit, the blocks are not required to be written back to the binary tree at the same time. Rather, they can be written back separately and in any order, as long as they are mapped to the same path. This flexibility is useful in designing different super

block algorithms.

4.2.3 Static Super Block

The above description of the super block scheme is very general and leaves many design decisions unspecified, for example, what size should a super block be, when and what blocks should be merged, etc.

Static super block has been proposed in previous work ([20]). In this scheme, every $n = 2^k$ data blocks consecutive in the program address space are merged into super blocks of size n. n is statically specified by the user before the program starts. n can be tuned for different applications or be the same for all applications. In the initialization stage of Path ORAM, blocks are merged into super blocks, each of which is forced to be mapped to the same path. During normal ORAM operations, a super block is accessed as a unit as described in Section 4.2.2.

Security

Similar to the argument of background eviction (Section 4.1.4), super block schemes are secure as long as a super block access is indistinguishable from a normal ORAM access. Security of normal Path ORAM is maintained since an access to a super block loads and remaps ***all*** blocks in the super block. Thus, subsequent accesses to this super block or to different super blocks will always be touching independently random paths. An adversary is not able to tell the super block size or whether the static super block scheme is used at all.

Limitations

Although the static super block scheme provides performance gain for programs with good locality, it has significant limitations which make it not practical:

First, it significantly hurts performance when the program has bad spatial locality (cf. Section 4.4). With these programs, prefetching always misses and pollutes the cache.

Second, it cannot adjust to different program behaviors in different phases. In practice, this leads to suboptimal performance for certain programs.

Third, it is the responsibility of programmers or the compiler to figure out whether the super block scheme should be used and the size of the super block.

4.3 Dynamic ORAM Prefetch: PrORAM

In this work, we propose a dynamic ORAM prefetching scheme called *dynamic super block*, which is the foundation of PrORAM, to address the limitations of the static scheme and make super block scheme practical. The dynamic super block scheme has the following key differences from the static scheme:

1. Crucially, super block merging is determined at runtime. Only blocks that exhibit spatial locality will be merged into super blocks. Programmers or compilers are not involved in this process.
2. In determining whether blocks should be merged into a super block, the dynamic super block scheme also takes into account the *ORAM access rate*, *prefetch hit rate*, etc. For example, if the prefetch hit rate is too low, merging should be stopped.
3. Finally, when a super block stops showing locality, the super block is broken. This makes it possible to adjust to program phases.

The dynamic super block scheme does not merge blocks during Path ORAM initialization. In other words, all blocks have $sbsize = 1$ after initialization. Accessing a block b in ORAM involves the following steps:

1. Access the path s where b is mapped to (according to the position map) and return to the processor's LLC (Last Level Cache) all the blocks that constitute the super block.
2. Super blocks are merged or broken according to spatial locality information.
3. Update the spatial locality statistics based on whether the prefetched blocks are used or not.

The second and third steps are what make the dynamic super block scheme different from the static scheme. We propose a *per block counter-based* scheme to efficiently measure spatial locality to guide block merging/breaking.

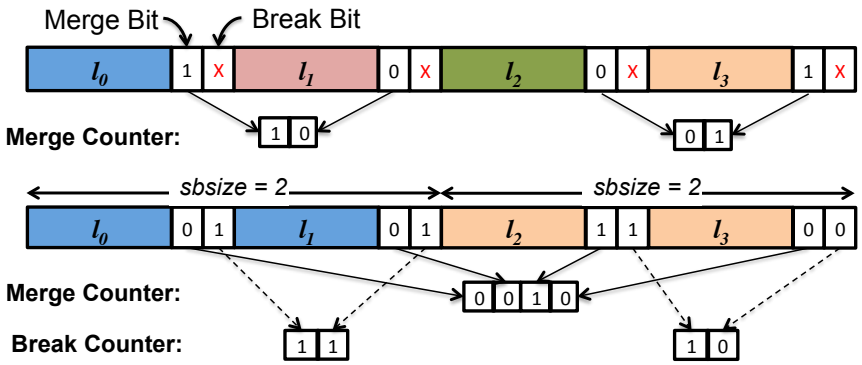


Figure 4.3.1: Hardware structure of *merge* and *break counter*. *Merge bits* from neighbor blocks form the their merge counter. Only super blocks have *break counters*.

4.3.1 Spatial Locality Counter

We first introduce the notion of a *neighbor block* to simplify the discussion. We call B' a neighbor block of block B, if they have the same size ($n = 2^k$) and can form a larger super block of size $2n = 2^{k+1}$. In Figure 4.2.2, block $0x02$ is a neighbor block of $0x03$ and super block $(0x00, 0x01)$ is a neighbor block of super block $(0x02, 0x03)$. However, super block $(0x02, 0x03)$ is not a neighbor block of super block $(0x04, 0x05)$. We restrict that only neighbor blocks can be merged into super blocks. Thus, as in the static super block scheme, only blocks with addresses differing in the last several bits can be merged into super blocks.

In order to decide what blocks should be merged, a *merge counter* and a *break counter* are introduced. Each pair of neighbor blocks have a *merge counter* indicating the spatial locality in them and to determine whether they should be merged or not (cf. Section 4.3.2). A *break counter* is associated with one super block to keep track of its spatial locality. A super block should be broken if it stops showing locality. The value of the counters will be updated based on the operations in Section 4.3.2 and Section 4.3.3.

Both counters are stored in the position map ORAM. Figure 4.3.1 shows the structure of a block in the position map ORAM (Pos-Map Block). Recall that each Pos-Map block contains the position map for multiple program addresses and stores the leaf labels for these addresses. A *merge bit* and a *break bit* are stored next to the leaf label in each position map. A counter is the concatenation of bits of the involved basic blocks. Once super blocks are merged or broken, the counters are reconstructed and the bits are reused for

different super block sizes. This keeps the hardware overhead small.

Note that the maximum super block size is limited by the maximum number of position maps stored in a Pos-Map block. With our parameters, each Pos-Map block (128B) stores 32 position maps (25 bits each) of consecutive addresses along with their merge and break bits (i.e., $32 \times (25+2) = 864$ bits). A super block in our scheme only consists of the basic blocks which are consecutive in the address space, and its size is always a power of 2. Therefore, position mapping of all the basic blocks of a super block always reside in the same Pos-Map block. When address a is accessed, the Pos-Map block containing a's mapping is loaded into the chip. The same Pos-Map block also contains the mappings of addresses near a (i.e., $\cdots, a-1, a+1, \cdots$) because they fit in the same Pos-Map block. As a result, whenever we access the position mapping of a [super]block, we also get the mapping for its neighbor blocks and its own sub-blocks along with their merge and break bits for free. Hence, there is no need for extra Pos-Map block accesses to load merge/break counter bits.

4.3.2 Merge Scheme

The merge operations are shown in Algorithm 4[1]. When a super block B of size n is returned to the LLC of the processor, the merge counter is first constructed. Since the position map for B and B' are stored in the same Pos-Map block, which has to be loaded into the ORAM controller before the data block can be accessed, the merge counter is completely reconstructed and can be operated on. The processor then detects whether all the n blocks in its *neighbor block* B' are also in the processor's cache. If so, we say B and B' have locality, and the merge counter of (B, B') is incremented. Otherwise, the merge counter will be decremented. If the merge counter reaches a *threshold*, B and B' are merged to a super block of size $2n$. How the *threshold* is determined will be discussed in Section 4.3.4.

Merging block B and B' is achieved by changing the position map of B to the position map of B'. (Note that B' is already in the cache before merging) The changes are written to the Pos-Map block.

The Pos-Map blocks also keep track of the super block size. When the Pos-Map block is loaded, if the corresponding blocks in it are mapped to the same leaf label, the ORAM controller then treats these blocks

[1] Incrementing a counter that is already the maximum value does not change the counter. Same for decrementing.

Algorithm 4 Merge Algorithm

```
Super block B is loaded from ORAM to LLC
Merge counter is constructed for B and its neighbor B'
if all blocks in B' are in LLC then
    (B, B').merge_counter ++
    if (B, B').merge_counter ≥ threshold then
        Merge B and B' into (B, B')
    end if
else
    (B, B').merge_counter - -
end if
```

as a super block.

Different from the static super block scheme discussed previously, Algorithm 4 dynamically exploits locality in the program. Blocks are merged only when they exhibit spatial locality, i.e., they are often present in the cache at the same time. After merging into super blocks, locality can be exploited since a single access now loads several useful data blocks.

4.3.3 Break Scheme

Break operations may happen when super blocks are accessed in the ORAM. This is the time when all the blocks in a super block B are on-chip and the break counter of B can be fully reconstructed. Each data block in the ORAM or LLC is associated with a *prefetch bit* and a *hit bit*. The *prefetch bit* indicates whether a basic block was prefetched. The *hit bit* indicates whether the block's last prefetch was used.

Algorithm 5 specifies the super block breaking algorithm. Without loss of generality, we assume that the interesting block is located in the first half of $B = (B_1, B_2)$, which is B_1. The algorithm starts with updating the break counter with prefetch/hit information of previous accesses. The break counter is incremented by one for a prefetch hit and decremented by one for a prefetch miss.

If the resulting break counter is smaller than a *threshold*, super block B will be broken. Breaking of B is done by remapping B_1 and B_2 to independent leaf labels. And the half that does not have the requested block (B_2 in our case) is written back to ORAM. Note that the position map for both B_1 and B_2 are on chip at this time and can be modified.

Algorithm 5 Break Algorithm

In ORAM controller

Super block $B = (B_1, B_2)$ is loaded from ORAM to LLC. The requested block is in B_1.
Reconstruct the break counter
for basic block b in B coming from ORAM **do**
    if b.prefetch **and not** b.hit **then**
        B.break_counter - -
    else if b.prefetch **and** b.hit **then**
        B.break_counter ++
    end if
    b.prefetch = false
end for
if B.break_counter ¡ threshold **then**
    break B into B_1 and B_2
    return B_1 to LLC and write B_2 back to ORAM
else
    for basic block b in B_2 **do**
        b.prefetch = true
        b.hit = false
    end for
end if

In Processor

when block b is accessed.
b.hit = true

Otherwise, the whole B will be returned to the LLC. In this case, each block in B_2 will have the *prefetch bit* set and *hit bit* reset indicating that the block is prefetched into the processor because B_2 is prefetched with respect to B_1 but has not been accessed yet. When a basic block with the *prefetch bit* set is accessed, a *prefetch hit* occurs and the *hit bit* is set. If a basic block has never been accessed since it was prefetched, the block will be evicted to ORAM with the *hit bit* unset; this is deemed a *prefetch miss*. Both the *prefetch bit* and the *hit bit* will be read the next time the super block is loaded.

4.3.4 Counter Threshold

For both the *merge counter* and the *break counter*, merge and break operations are carried out when the value of the counter reaches a threshold. Properly determining the threshold value is important in achieving good system performance. We provide two algorithms to determine the threshold: *static thresholding* and *adaptive thresholding*.

Static Thresholding

Static thresholding is very simple. The initial value of the merge counter is set to 0. Two neighbor blocks B_1 and B_2 of size $n = 2^k$ are merged when the value of their merge counter is higher or equal to $2n$ (note that this threshold fits in the merge counter which is $2n$ bits long). For block size of 1, 2 and 4 before merging, this corresponds to the threshold value of 2, 4 and 8, respectively. The threshold increases for larger block sizes because larger blocks incur more dummy accesses which may hurt performance.

Similarly, the initial value of break counter is $2n$ where n is the super block size. In our scheme, the threshold of break counter is 0, which is the minimal value of the break counter.

Adaptive Thresholding

With static thresholding, blocks are merged whenever they exhibit enough data locality. However, even if all blocks have perfect spatial locality, if too many blocks are merged into large super blocks, system performance would still suffer due to the large number of background evictions required to ensure that the

stash does not overflow (cf. Section 4.4.5). We propose to use *adaptive thresholding* to resolve this problem.

In particular, we propose to use the following equation to calculate the threshold.

$$threshold = C \times \frac{sbsize^2 \times eviction\_rate \times access\_rate}{prefetch\_hit\_rate} \tag{4.3.1}$$

eviction_rate is the number of background evictions divided by the total number of memory requests. *access_rate* is the percentage of time when the ORAM is busy. *prefetch_hit_rate* is the percentage of hits out of all prefetched blocks. These numbers are collected within a time window and be updated periodically (every 1000 ORAM requests in this work). Note that a larger threshold makes it harder to merge into super blocks.

The intuition behind the equation is fairly simple. As the threshold goes up, less blocks would be merged into super blocks, which reduces the number of background evictions. Take merging threshold as an example—when *sbsize* is large, we want to raise the threshold to be conservative [2] since larger *sbsize* incurs more background evictions. When *eviction_rate* and *access_rate* are high, we raise the threshold to prevent further increasing of background eviction. The *prefetch_hit_rate* is the opposite: we want to lower the threshold and merge more blocks when *prefetch_hit_rate* is high, which means block merging is accurate. The same arguments also hold for break threshold.

In practice, the merge threshold and break threshold are different by a small term to introduce some hysteresis into the system. In particular, $threshold_{Merge} = threshold + sbsize$ and $threshold_{Break} = threshold$. This prevents the case where a block keeps being merged and broken constantly.

Notice that the equation is not provably the optimal equation for the merge/break threshold, but it is simple and easy to implement in hardware. We leave the exploration of more complicated thresholding algorithms to future work.

4.3.5 Hardware Support

The *hit bit* is stored with each data block in the ORAM and the LLC, since it is updated on an LLC hit and the corresponding position map block may not be on-chip. The *merge bit*, *break bit* and the *prefetch bit* are

[2] Experimental results show that $sbsize^2$ performs better than *sbsize*.

stored in the Pos-Map blocks. We assume 128-byte block size, so the storage overhead of dynamic super block is only 4 bits per block, less than 0.4%.

For the merging scheme, when a block B is loaded into the LLC, we need to probe the LLC to check if the neighbor block B' exists in the cache (cf. Section 4.3.2). Only the tag array of the LLC needs to be accessed for this purpose. This can be done in parallel with the ORAM access and is not on the critical path. For the breaking scheme, the break counter is updated based on the prefetch bit and hit bit of each block in B, using simple comparison and arithmetic operations. When a block is accessed in LLC, the hit bit needs to be set. In summary, the scheme only involves several cache lookups and simple operations. They are cheap compared to the path read/write and data encryption/decryption in an ORAM access.

4.3.6 Security of Dynamic Super Block

The threat model under discussion is identical to prior ORAM work. We claim that ORAM with a dynamic super block scheme maintains the same level of security as a normal ORAM. In other words, adding dynamic super blocks to ORAM does not change the security guarantee of the original ORAM.

Following the security of the static super block scheme, accessing a super block of any size will look indistinguishable from accessing a normal data block because all the blocks in a super block are remapped at the same time. To demonstrate the security of dynamic super block, we only need to show the security of merging and breaking processes.

For merging, assume that block B_1 and B_2 (mapped to leaf s_1, s_2 respectively) are merged into a super block $B = (B_1, B_2)$. After merging, both blocks are mapped to a same leaf s which is a new independent random number, and is unlinkable to s_1, s_2 or any other previously accessed path. From the adversary's point of view, the leaves that are accessed in the ORAM are not linkable to each other. The adversary cannot figure out whether merging happens in an ORAM access at all.

Similarly, if block $B = (B_1, B_2)$ (mapped to s) breaks, the two halves B_1 and B_2 (mapped to s_1 and s_2) will be mapped to two independent random leaves. When one of B_1 and B_2 is accessed later, the leaf being accessed will be unlinkable to the leaf of the other half or s. This indicates that when or whether breaking happens cannot be learned by observing the ORAM access sequence.

To this point, the dynamic super block scheme does not leak any more information through the access pattern. However, one may argue that super block schemes leak *locality* information through timing channels. For example, merging blocks into super blocks reduces the total number of ORAM accesses which may be an indicator that the program has good spatial locality.

Although it is true that locality information may be learned through timing attacks, as said in Section 4.1.5, timing protection is not part of the original ORAM definition [18]. Very few ORAM designs in the literature considered timing attacks (i.e., when ORAM accesses happen or the total number of accesses) and ORAMs in general break under timing attacks. In order to protect the timing channel, periodic ORAM accesses need to be adopted ([10]), which can be easily added on top of ORAM with super blocks. We evaluate this design point in Section 4.4.6.

With periodic ORAM accesses to prevent timing channel leakage, the ORAM accesses are completely deterministic during the normal execution of a program and no leakage can happen. The only possible timing leakage is through the program's total execution time. As discussed by Fletcher *et al.* [62], the execution time of a program only leaks $\lg(T)$ bits information where T is the number of cycles the program takes. This is a very small leakage and applies to all types of ORAMs; adding a super block mechanism does not change it. Although it is true that enabling super blocks on a system may change the total runtime which tells the adversary some information about data locality, the same argument applies to other system components as well. For example, enabling vs. disabling branch prediction or the L3 cache leaks the branch behavior or memory footprint of the program. Super block is just one of these components and does not leak more information than other components.

To conclude, the dynamic super block scheme or any super block scheme in general maintains the same security level as conventional ORAMs. No extra leakage is introduced.

Table 4.4.1: System Configuration.

Secure processor configuration	
Core model	1 GHz, in order core
L1 I/D Cache	32 KB, 4-way
Shared L2 cache	512 KB per tile, 8-way
Cacheline (block) size	128bytes
DRAM bandwidth	16 GB/s
conventional DRAM latency	100 cycles
Default ORAM configuration	
ORAM Capacity	8 GB
Number of ORAM hierarchies	4
ORAM basic block size	128 Bytes
Path ORAM latency	2364 cycles
Z	3
Max Super Block Size	2
Stash Size	100

4.4 Evaluation

4.4.1 Methodology

Graphite [63] is used as the simulator in our experiments. Graphite simulates a tiled multi-core chip. The hardware configurations are listed in Table 4.4.1. We assume there is only one memory controller on the chip. While the insecure DRAM model can exploit bank-level parallelism and issue multiple memory requests at the same time (according to the Graphite DRAM model), all ORAM accesses are serialized (cf. Section 4.1.6).

The DRAM in Graphite is simply modeled by a flat latency. The 16 GB/s is calculated assuming a 1 GHz chip with 128 pins and pins are the bottleneck of the data transfer. Although this model neglects many DRAM internal structures, previous work ([20]) showed that using a more accurate model does not change the result much.

We use Splash-2 [31], SPEC06 [32] and a database management system (DBMS) application [33] to evaluate different ORAM prefetching techniques. For DBMS, we run two OLTP benchmarks: YCSB [64]

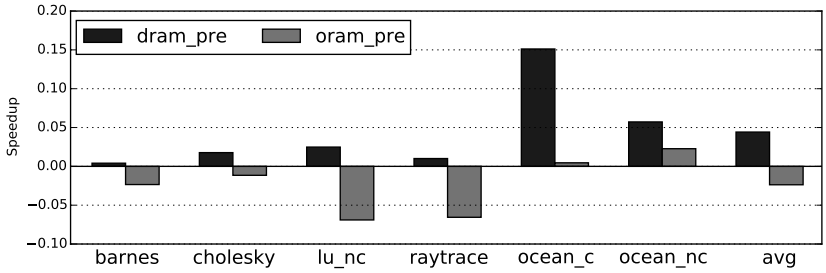


Figure 4.4.1: Traditional data prefetching on DRAM and ORAM

and TPCC [65]. For most of the experiments, we will show both the speedup and the total memory accesses with respect to the baseline ORAM. The number of memory accesses is proportional to the energy consumption of the memory subsystem.

Three baseline designs are used for comparison: the insecure baseline using normal DRAM, the baseline Path ORAM without super blocks (*oram*) and the static super block scheme (*stat*). The default parameters for ORAM are shown in Table 4.4.1. Unless otherwise stated, all the experiments use these ORAM parameters.

We first evaluate a traditional stream prefetcher on both DRAM and Path ORAM (Section 4.4.2). Then, we will show the performance of different super block schemes with both synthetic and real benchmarks (Section 4.4.4). Different system parameters are explored in the sensitivity study section (Section 4.4.5). Finally, we evaluate the impact of having periodic ORAM accesses on these algorithms (Section 4.4.6).

4.4.2 Traditional Prefetching on Path ORAM

As discussed in Section 4.2.1, traditional prefetching does not help much in the context of ORAM. The reason behind this is the available memory bandwidth budget. Traditional DRAM prefetchers utilize the available bandwidth between useful accesses to issue prefetch requests. However, ORAM does not have that extra bandwidth available to issue prefetch requests. This conclusion is verified in Figure 4.4.1. Here, prefetching is added to both DRAM and ORAM based systems. Prefetching helps to improve performance on DRAM based systems. The ORAM, however, takes too much memory bandwidth and the memory subsystem is busy serving useful requests. Inserting prefetch requests will delay these useful requests. In

the best case, if the prefetch hits and the prefetching is timely, the performance does not suffer. But if any prefetch misses, the performance decreases.

4.4.3 Synthetic Benchmark

In this section, a synthetic benchmark is used to study different aspects of super block schemes. $Z = 4$ is chosen here to make it easier to see the performance difference. The synthetic benchmark accesses an array with two patterns, *sequential* or *random*. For the *sequential* pattern, the part of the array is scanned sequentially, leading to good spatial locality. For the *random* pattern, the data is randomly accessed with no spatial locality.

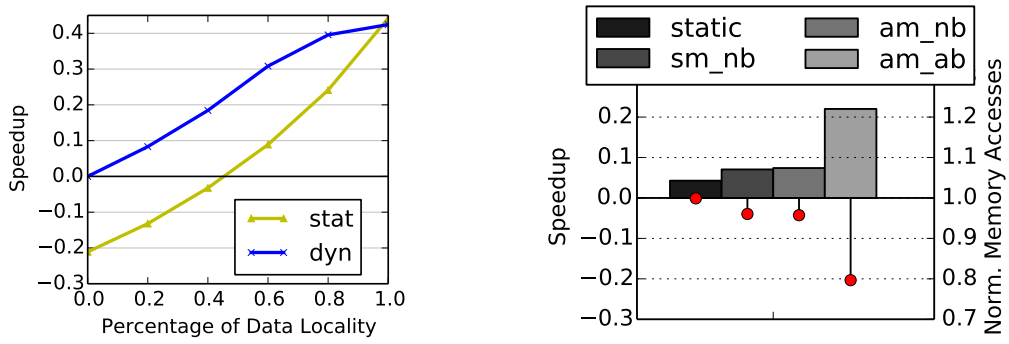
Locality

In Figure 4.4.2a, we sweep the percentage of data with locality. A benchmark with X% locality means that X% percentage of data are accessed sequentially and the rest are accessed randomly. Only the sequentially-accessed data has locality and can benefit from having super blocks.

As the figure shows, the static super block scheme only works when there is good spatial locality in the application and performs worse than the baseline ORAM if locality is bad. The dynamic super block scheme, on the other hand, always outperforms both the baseline ORAM and the static super block scheme. When there is no locality at all, the dynamic super block scheme has the same performance as the baseline ORAM. As the locality increases, performance also increases. Finally, the dynamic super block scheme has similar performance as the static scheme when there is 100% locality.

Phase Change

The benchmark in Figure 4.4.2a only has static locality behavior, namely, the part of the data having locality always has locality throughout the whole execution. In practice, many programs have phase change behavior. Figure 4.4.2b models the phase change behavior where different parts of the data exhibit locality in different phases. Specifically, in the first phase, half of the data are accessed sequentially and the other half randomly.



(a) Locality Sweep

(b) Phase Change

Figure 4.4.2: Different locality in the synthetic benchmark. Speedup is measured with respect to the baseline ORAM.

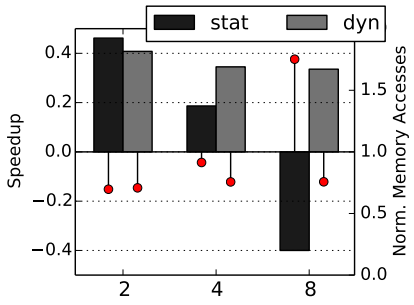


Figure 4.4.3: Sweep super block size of synthetic benchmark

In the second phase, the first (second) half is randomly (sequentially) accessed. The pattern keeps switching in the following phases of the program.

In Figure 4.4.2b, *Sm*,*Am* stands for static and adaptive merging and *Nb*,*Ab* stands for no breaking and adaptive breaking respectively, and *static* represents the static super block scheme. In this case, it is obvious that breaking helps improve the performance. In the phases with locality, blocks will be merged to improve performance. And in the phases without locality, super blocks will be broken to prevent inaccurate prefetching. This helps to reduce the number of background evictions and improve prefetch hit rate.

Super Block Size

Figure 4.4.3 sweeps the super block size (*sbsize*) in different super block schemes (for dynamic super block, *sbsize* is the maximum super block size). We run the synthetic benchmark which has 100% spatial locality.

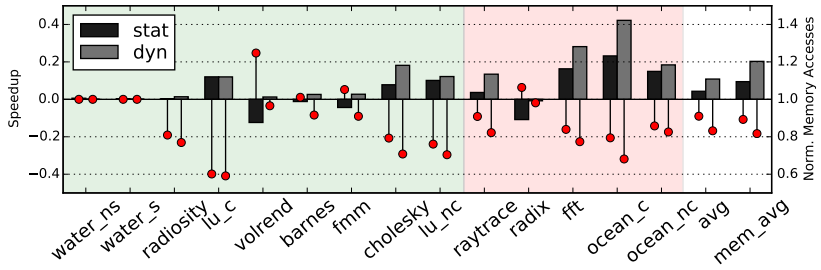
Even with perfect locality, as *sbsize* increases, performance of the static super block scheme still degrades quickly due to excessive background evictions. On the other hand, the dynamic super block scheme will throttle merging of too large super blocks using the *adaptive thresholding* strategy introduced in Section 4.3.4. Once the background eviction rate is too high, super block merging is stopped and background eviction rate is kept low.

4.4.4 Real Benchmarks

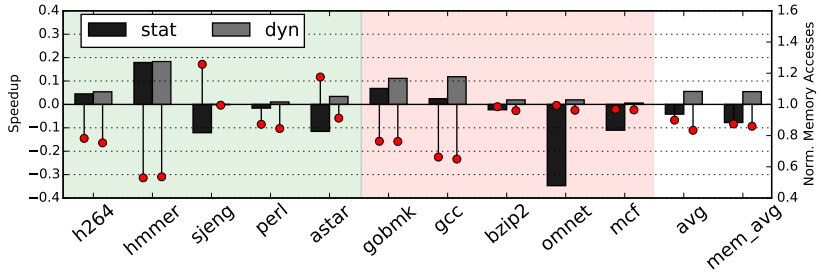
Even though Path ORAM has inherent performance overhead over DRAM as explained in Section 4.1.6, it is important to note that this overhead is proportional to the memory intensiveness of the application. Memory intensive applications have significantly higher overhead than computation intensive applications. Figure 4.4.4a and Figure 4.4.4b show Splash2 and SPEC06 benchmarks respectively sorted in ascending order with respect to the overhead of baseline ORAM over DRAM. We consider all the benchmarks with less than 2× overhead as *Computation Intensive* benchmarks (plotted over green background) and all those with more than 2× overhead as *Memory Intensive* benchmarks (plotted over red background).

As pointed out in Section 4.2.3, the static super block scheme is very sensitive to the nature of the application. I.e., it only shows performance gain for benchmarks that have good spatial locality (e.g., *ocean_contiguous*) which is clear from Figure 4.4.4. On some benchmarks (e.g., *volrend, radix, sjeng, astar, omnet, mcf, TPCC*), the static super block scheme gets worse performance than the baseline ORAM. This is either due to the fact that these benchmarks lack locality or that excessive background evictions hurt performance.

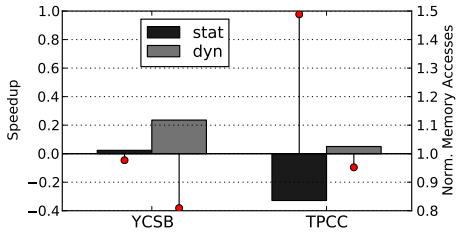
On the other hand, the dynamic super block scheme outperforms the baseline ORAM on all the benchmarks we evaluate here. On average, the performance gain of the dynamic super block scheme for memory intensive Splash2 benchmarks (*mem_avg*) over baseline ORAM is 20.2% whereas the overall average gain



(a) Splash2



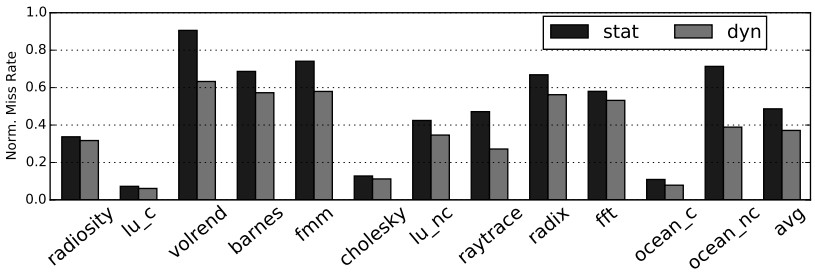
(b) SPEC06



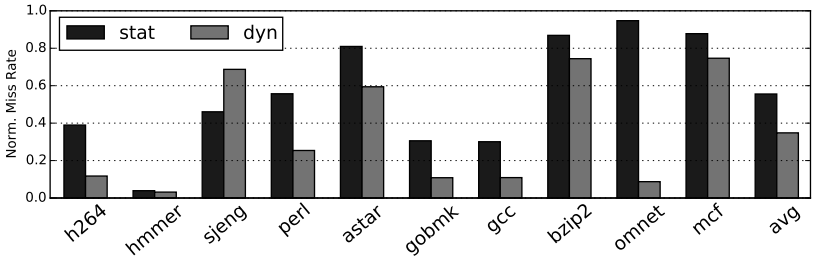
(c) DBMS

Figure 4.4.4: Speedup and normalized memory access count (with respect to baseline ORAM) of static and dynamic super block schemes on Splash2, SPEC06 and DBMS benchmarks.

(*avg*) is 10.6% i.e., twice as much than what the static super block scheme offers. The overall average performance gain over the baseline ORAM for SPEC06 benchmarks is 5.5% whereas for DBMS, the performance gain over the baseline ORAM is 23.6% for YCSB and 5% for TPCC. The performance gain is most prominent for highly memory bound benchmarks. For *ocean_contiguous* for example, the performance gain is 42%. We also show the total number of ORAM accesses (normalized to the baseline ORAM) in Figure 4.4.4 (shown by red markers). This is proportional to the energy consumption of the memory subsystem. On average, the dynamic super block scheme saves 16.8% energy for Splash2, 16.6% for SPEC06



(a) Splash2



(b) SPEC06

Figure 4.4.5: Miss rate for different Path ORAM schemes on Splash-2 and SPEC06 benchmarks.

and 19.1% (4.8%) for YCSB (TPCC) over plain ORAM.

Figure 4.4.5 shows the prefetch miss rates of static and dynamic super block schemes on Splash2 and SPEC06 benchmarks. *water-spatial* and *water-nsquared* are not shown here since they are too compute bound and do not access ORAM frequently. Since the static super block scheme prefetches all the neighbor blocks, the miss rate is very high for benchmarks that lack spatial locality (e.g., *volrend, omnet*). On average, the dynamic super block scheme lowers the overall prefetch miss rate of static super block from 48.6% to 37.1% for Splash2 benchmarks and from 55.5% to 34.8% for SPEC06 benchmarks.

4.4.5 Sensitivity Study

In this section, we will study how different parameters in the system affect the performance of super block schemes.

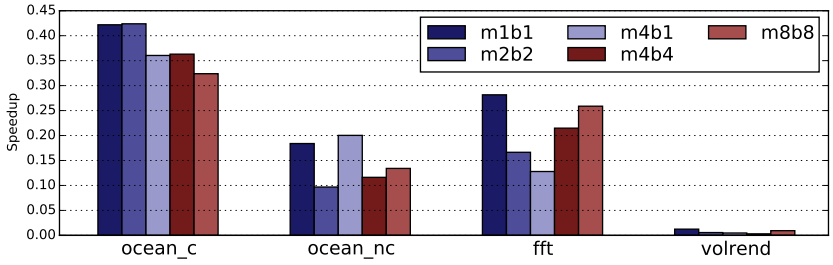


Figure 4.4.6: Sweep the coefficient in merging and breaking strategies.

Merge/Break coefficient

Equation 4.3.1 in Section 4.3.4 has a coefficient C unspecified. We now study how the coefficient affects the performance of super blocks.

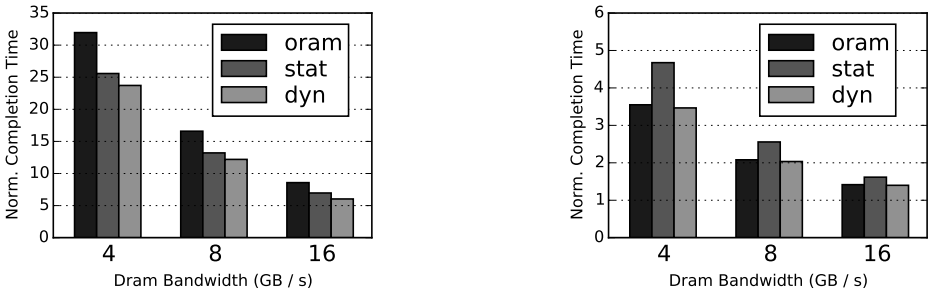
Figure 4.4.6 sweeps the merge and break coefficients (C_{merge} and C_{break}). mxny in the figure means that $C_{merge} = x$ and $C_{break} = y$.

For benchmarks with good spacial locality (e.g., *ocean_contiguous*, *ocean_non_contiguous*), smaller coefficient makes it easier for blocks to be merged into super blocks. As a result, merging happens earlier in the execution leading to better performance. For benchmarks with bad spacial locality (e.g., *volrend*), the coefficient actually does not have an effect on the performance, because merging does not happen regardless of the value of the coefficient.

For the rest of the work, we use $C_{merge} = C_{break} = 1$ for our experiments.

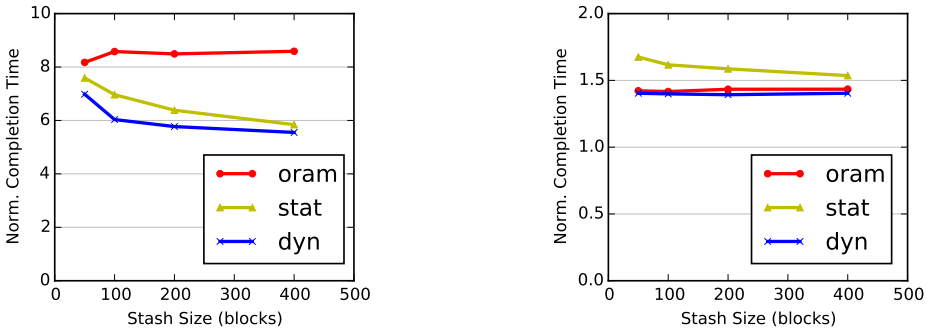
DRAM Bandwidth

The DRAM has a default bandwidth of 16 GB/s in our evaluations, which is calculated assuming that the pin bandwidth is the system bottleneck. In practice, however, this bandwidth might not be achievable. In Figure 4.4.7, we sweep different DRAM bandwidth values and the performance gain of the dynamic super block scheme is consistent across all configurations for memory intensive benchmarks (Figure 4.4.7a). This is because super blocks improve the memory efficiency in general by reducing the total number of ORAM accesses. This gain is orthogonal to the DRAM bandwidth.



(a) ocean_contiguous (b) volrend

Figure 4.4.7: Sweep DRAM bandwidth.



(a) ocean_contiguous (b) volrend

Figure 4.4.8: Sweep stash size.

For benchmarks with little or no locality (Figure 4.4.7b), the dynamic super block scheme does not have much gain over the baseline ORAM since merging does not take place. But both the dynamic super block scheme and the baseline ORAM have performance gain over the static super block scheme where blocks are blindly merged resulting in a large number of cache misses.

Stash Size

As discussed in Section 4.1.2, the stash is an on-chip data structure which temporarily holds the blocks that cannot be evicted back to the binary tree. Whenever the stash becomes full, the ORAM does background eviction which introduces an extra ORAM access doing no real work. A larger stash is less likely to become

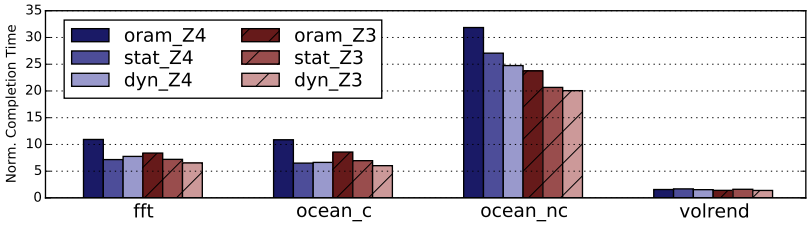


Figure 4.4.9: Different Z values.

full and thus reduces background eviction rate and improves performance.

In Figure 4.4.8, the stash size is swept for two different benchmarks, one with good spatial locality (*ocean_contiguous*) and the other with bad spatial locality (*volrend*). For both benchmarks, the performance of baseline ORAM does not change much when the stash size increases. This is because that the baseline ORAM already has a very low background eviction rate and enlarging the stash only gives marginal gain.

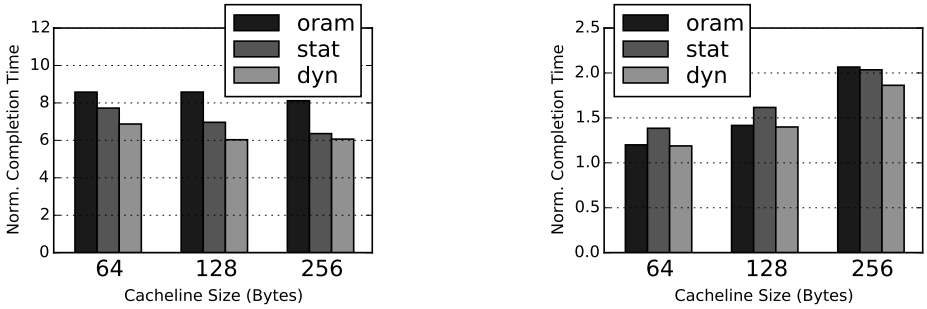
For super block schemes, however, background eviction rate is high because multiple blocks may be added to the stash in each ORAM access. As a result, the performance increases as stash size becomes larger. For *ocean_contiguous*, both the dynamic and static super block schemes gain from larger stash size. For *volrend*, only the static super block scheme merges blocks into super blocks.

In general, dynamic super block scheme shows significant performance gain over ORAM even at small stash sizes in contrast to static super block.

Z Value

Having larger Z increases the latency of each ORAM access since more data needs to be loaded. On the other hand, having smaller Z increases the background eviction rate. Previous work [20] showed that $Z = 3$ provides the best performance without super block. It is also the default parameter setting in this work.

In Figure 4.4.9, both $Z = 3$ and $Z = 4$ are evaluated for baseline ORAM, static and dynamic super block schemes. $Z = 3$ achieves better performance than $Z = 4$ for the baseline ORAM, which corroborates previous results. The dynamic super block scheme has consistent performance gain for both Z values.



(a) ocean_contiguous

(b) volrend

Figure 4.4.10: Sweep cacheline size.

Cacheline Size

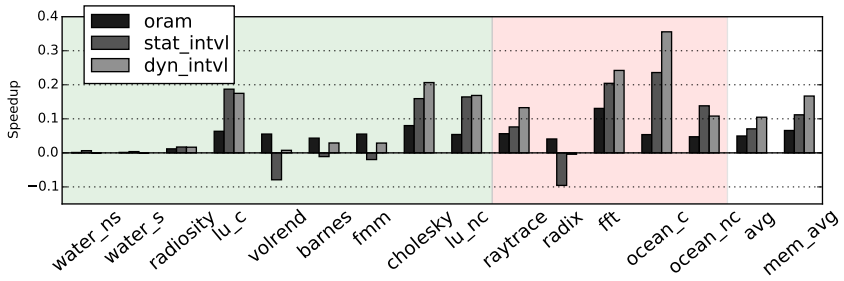
The default cacheline size is 128 Bytes in this work, in order to match the parameters in the previous work ([20]). Figure 4.4.10 shows the performance of different ORAM schemes at three different cacheline sizes (64, 128 and 256 Bytes). In general, the behaviors of dynamic and static super block schemes do not change.

4.4.6 Protecting Timing Channel

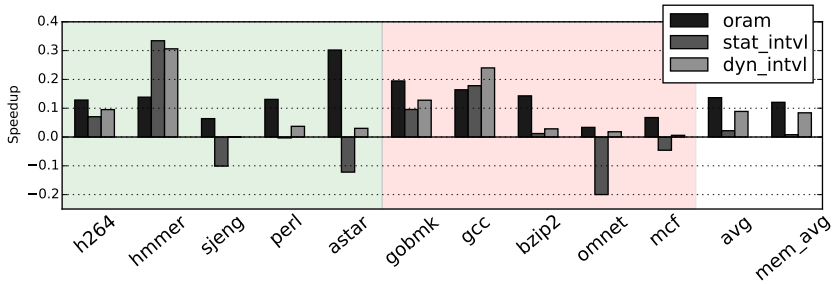
As pointed out in Section 4.3.6, the ORAM definition does not try to protect timing attacks. And an adversary may still learn lots of information from the timing of memory accesses. We need periodic ORAM accesses to protect the timing channel. Both dynamic and static super block schemes can be easily integrated with periodic ORAM accesses. Figure 4.4.11 shows the simulation results of periodic ORAM with static super blocks (*stat_intvl*) and dynamic super blocks (*dyn_intvl*) normalized to periodic ORAM. For comparison, non-periodic ORAM (*oram*) results are also shown. O_{int} is defined as the number of cycles between two consecutive ORAM accesses, which is chosen to be 100 cycles in this experiment.

Two observations can be made from Figure 4.4.11.

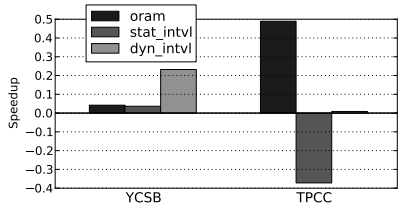
First, for most applications, adding periodicity in ORAM accesses does not significantly hurt performance. On average, only 3.6% additional performance degradation is incurred for Splash2. Part of the reason is that the O_{int} is chosen to be small in our evaluations thus ORAM bandwidth is almost maximized.



(a) Splash2



(b) SPEC06



(c) DBMS

Figure 4.4.11: Periodic ORAM accesses. Speedup is respect to the baseline ORAM with periodic accesses. O_{int} = 100 cycles.

Second, dynamic super blocks provide performance gain regardless whether periodicity is used or not and therefore, once integrated with periodic ORAM, it can still provide significant speedup while preventing the information leakage over the timing channel.

Since the ORAM has a strictly periodic access pattern, the energy consumption of different ORAM schemes would be the same. However, this performance advantage of dynamic super blocks can be easily translated to energy advantage by setting O_{int} high, which slows down the system but makes it more energy efficient ([62]).

4.5 Related Work

This work mainly focuses on applying data locality optimizations to *Oblivious RAM*. The most relevant previous works are ORAM optimization techniques and locality optimizations in the memory system.

4.5.1 ORAM Optimization

Previous work [20] has explored the Path ORAM design space and proposed a static super block scheme. We used this optimized Path ORAM as the baseline. We extend the static super block scheme to a dynamic super block scheme which is significantly more stable and has better performance.

Previous work has exploited the fact that ORAM operations can be easily parallelized and assigned to multiple trusted coprocessors [66]. The optimization techniques proposed in this work are orthogonal and can be applied on top of [66].

While we used Path ORAM for discussion, the optimization techniques proposed are not constrained to Path ORAM. For example, other ORAM schemes (e.g., [56]) have similar binary tree structure to Path ORAM. After adding background eviction, these ORAM schemes can also benefit from using super blocks. In general, all ORAM schemes should be able to take advantage of super blocks as long as they have support for background eviction.

4.5.2 Exploiting Locality in Memory

In the architecture community, there has been lots of work exploiting data locality in programs. An important technique that has been widely used is data prefetch [67, 68, 69, 70], where the processor loads blocks that are likely to be accessed in the near future into the cache.

In this work, we showed that traditional prefetching techniques do not work well in an ORAM context, and that super block schemes take advantage of ORAM internal structure to exploit locality.

Our work makes the assumption that only the blocks consecutive in address space can be merged into super blocks. However, previous work in data prefetch [69] allows data striding in the address space to be prefetched. Merging striding blocks is also possible for the dynamic super block scheme. Such exploration

is left for future work.

4.6 Chapter Review

A novel ORAM prefetcher: *PrORAM* based on *dynamic super block* is proposed in this chapter. The implementation details are discussed and the design space is comprehensively explored. PrORAM introduces the first practical super block scheme and is much more stable than a static super block scheme over different benchmarks. On memory intensive Splash-2 and SPEC06 benchmarks, PrORAM improves over baseline ORAM by 20.2% and 5.5% in terms of completion time and reduces energy consumption by 16.8% and 16.6% respectively. For DBMS, the performance gain is 23.6% and 5% and energy reduction is 19.1% and 4.8% for YCBS and TPCC respectively.

Chapter 5

Privacy Leakage via Write-Access Patterns to the Main Memory

As discussed in Chapter 4, users' data privacy is becoming a major concern in computation outsourcing in the current cloud computing world. Numerous secure processor architectures (e.g., XOM [5, 6], TPM+TXT [4], Aegis [8], Intel-SGX [12] etc.) have been proposed for preserving data confidentiality and integrity during a remote secure computation. While the secure processors provide sufficient levels of security against *direct* attacks, most of these architectures are still vulnerable to *side-channel* attacks. For instance, XOM and Aegis architectures are vulnerable to control flow leakage via address bus snooping [71, 72, 73]. Similarly, Intel-SGX, being a strong candidate in secure architectures, is vulnerable to side-channel attacks via a compromised OS[74].

Zhuang *et al.* [75] showed that although the data in the main memory of the system can be encrypted, the access patterns to the memory could still leak privacy. An adversary who is able to monitor both read *and* write accesses made by an application can relate this pattern to infer secret information of the application. For example, Islam *et al.* [76] demonstrated that by observing accesses to an encrypted email repository, an adversary can infer as much as 80% of the search queries. This, however, is a very strong adversarial model which, in most cases, requires direct physical access to the memory address bus. In cloud computing, for example, this requires the cloud service itself to be untrusted. The challenging requirements posed by

the above mentioned strong adversarial model leads one to think that applications are vulnerable to privacy leakage via memory access patterns only if such a strong adversary exists, i.e., one capable of monitoring both read *and* write accesses.

In this chapter, we counter this notion by demonstrating privacy leakage under a significantly weaker adversarial model. In particular, we show that an adversary capable of monitoring *only* the write access patterns of an application can still learn a significant amount of its sensitive information. Hence, in the model of computation outsourcing to a secure processor discussed earlier, even if the cloud service itself is trusted, a remote adversary is still able to steal private information if the underlying hardware does not protect against leakage from write access patterns.

We present a real attack on the famous Montgomery's ladder technique [77] commonly used in public key cryptography for modular exponentiation. Exponentiation algorithms, in general, are vulnerable to various timing and power side-channel attacks [78, 79, 80]. Montgomery's ladder performs redundant computations as a countermeasure against power side-channel attacks (e.g., simple power analysis [81]). However, by monitoring the order of write accesses made by this algorithm, one can still infer the secret *exponent* bits.

In our weaker adversarial model, since we cannot directly monitor the memory address bus, we learn the pattern of write accesses by taking frequent memory snapshots. For this purpose, we exploit a compromised Direct Memory Access device (DMA[1]) attached to the victim computer system to read the application's address space in the system memory [82, 83, 84]. Clearly, any two memory snapshots only differ in the locations where the data has been modified in the latter snapshot. In other words, comparing the memory snapshots not only reveals the fact that write accesses (if any) have been made to the memory, but it also reveals the exact locations of the accesses which leads to a precise access pattern of memory writes.

Our experimental setup uses a PCI Express to USB 3.0 adapter attached to the victim system, alongside an open source application called PCILeech [85], as the compromised DMA device. We implement the Montgomery's ladder for exponentiation of a 128 byte message with a 64 byte (512 bits) secret exponent [86]. Through our attack methodology, we are able to infer all 512 secret bits of the exponent in just 3

[1]DMA grants full access of the main memory to certain peripheral buses, e.g. FireWire, Thunderbolt etc.

Algorithm 6 RSA - Left-to-Right Binary Algorithm

Inputs: $g, k = (k_{t-1}, \cdots, k_0)_2$ **Output:** $y = g^k$

Start:

1: $R_0 \leftarrow 1; R_1 \leftarrow g$
2: **for** $j = t-1$ *downto* 0 **do**
3: $\quad R_0 \leftarrow (R_0)^2$
4: $\quad$ **if** $k_j = 1$ **then** $R_0 \leftarrow R_0 R_1$ **end if**
5: **end for**

return R_0

minutes and 34 seconds on average.

Although our experimental setup utilizes a wired connection to a USB 3.0 port on the victim system for DMA, Stewin *et al.* demonstrated that DMA attacks can also be launched *remotely* by injecting malware to the dedicated hardware devices, such as graphic processors and network interface cards, attached to the host platform [87]. Therefore, our attack methodology allows even remote adversaries to exploit the coarse grained side-channel information obtained by memory snapshots to infer the secret data. Hence, this effort opens up new research avenues to explore efficient countermeasures to prevent privacy leakage under remote secure computation.

5.1 Background

5.1.1 Exponentiation Algorithms

Exponentiation algorithms have central importance in cryptography, and are considered to be the backbone of nearly all the public-key cryptosystems. Although numerous exponentiation algorithms have been devised, algorithms for constrained devices are scarcely restricted to the square-and-multiply algorithms. RSA algorithm, used in e.g. Diffie-Hellman key agreement, is a commonly used exponentiation algorithm which performs computation of the form $y = g^k \mod n$, where the attacker's goal is to find the secret key k. The commonly used square-and-multiply implementation of this algorithm is shown in Algorithm 6. For a given input g and a secret key k, Algorithm 6 performs multiplication and squaring operations on two local variables R_0 and R_1 for each bit of k starting from the most significant bit down to the least significant bit.

Algorithm 7 Montgomery Power Ladder Algorithm

Inputs: $g, k = (k_{t-1}, \cdots, k_0)_2$ **Output:** $y = g^k$
Start:

1: $R_0 \leftarrow 1; R_1 \leftarrow g$
2: **for** $j = t-1$ *downto* 0 **do**
3: **if** $k_j = 0$ **then** $R_1 \leftarrow R_0 R_1;\ \ R_0 \leftarrow (R_0)^2$
4: **else** $R_0 \leftarrow R_0 R_1;\ \ R_1 \leftarrow (R_1)^2$
5: **end if**
6: **end for**

return R_0

Notice that the conditional statement on line 4 of Algorithm 6 executes based on the value of secret bit k_j. Such conditional branches result in two different power and timing spectra of the system for $k_j = 0$ and $k_j = 1$, hence leaking the secret key k over the timing/power side-channels. Similar attacks [86] have leaked 508 out of 512 bits of an RSA key by using branch prediction analysis (BPA). Thus, the attack-prone nature of RSA algorithm (Algorithm 6) poses a need for an alternate secure algorithm.

5.1.2 Montgomery's Power Ladder Algorithm

Montgomery Power Ladder [77] shown in Algorithm 7 performs exponentiation without leaking any information over power side-channel. Regardless of the value of bit k_j, it performs the same number of operations in the same order, hence producing the same power footprint for $k_j = 0$ and $k_j = 1$. Notice, however, that the specific order in which R_0 and R_1 are updated in time depends upon the value of k_j. E.g., for $k_j = 0$, R_1 is written first and then R_0 is updated; whereas for $k_j = 1$ the updates are done in the reverse order. This sequence of write access to R_0 and R_1 reveals to the adversary the exact bit values of k. In this work, we exploit this vulnerability in a real implementation of Montgomery ladder to learn the secret key k.

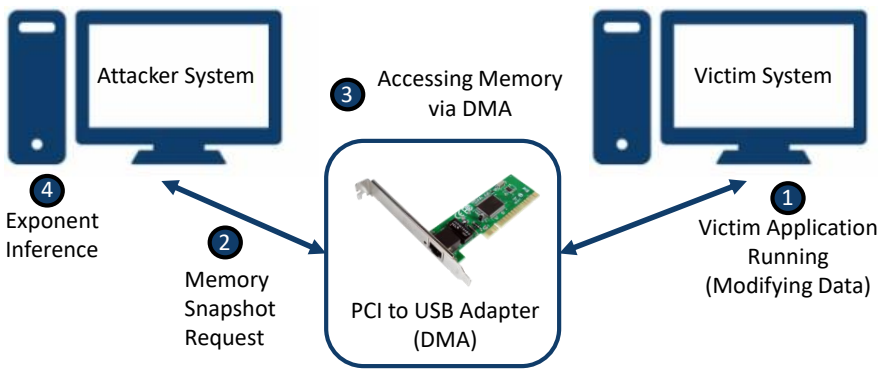


Figure 5.2.1: Our adversarial model: The attacker system takes snapshots of the victim's DRAM via the PCI adapter to infer the secret key.

5.2 The Proposed Attack

5.2.1 Adversarial Model

Consider a computer system that is continuously computing exponentiations of the form $y = g_i^k$ for the given inputs g_i using the same secret exponent k according to Algorithm 7. We call this system the *victim* system. All the data stored in the main memory of this system is encrypted. Let there be a compromised DMA device (e.g., a PCI-to-USB adapter) connected to the victim system through which an attacker system can read the whole main memory of the victim as shown in Figure 5.2.1. The attacker system, however, is limited in its ability to successively read the victim's memory by the data transfer rate of the underlying DMA interface. The adversary's goal is to find the key k by learning the application's write pattern through frequent snapshots of the victim system's memory. The victim system used in our attack comes with a *write-through* cache configuration enabled by default. As a result, any write operations performed by the application are immediately propagated through the memory hierarchy down to the untrusted DRAM. Furthermore, we assume that the victim application receives all the inputs g_i in a batch and continuously produces the corresponding cipher texts such that the physical memory region allocated to the application during successive encryptions remains the same. In other words, the application is not relocated to a different physical address space by the OS throughout the attack. Such use cases can be found in the applications

Algorithm 8 Victim App's Address Space Identification

Inputs: M: Set of memory blocks to scan.
Output: S: Set of application's memory block(s).
Start:

```
1: S = ∅                                          ▷ Initially empty set.
2: for m ∈ M do                                   ▷ Scan each block.
3:     s1 = TAKESNAPSHOT(m)
4:     s2 = TAKESNAPSHOT(m)
5:     if COMPAREMATCH(s1, s2) then
6:         S = S ∪ m
7:     end if
8: end for
9: return S
```

that require computing signatures of large files.

5.2.2 Attack Outline

Given the above mentioned setting, we proceed with our attack methodology as follows: First, a full scan of the victim's memory is performed to identify the physical address space allocated to the victim's application. Since the adversary requires victim application's memory snapshots at a high frequency, it is infeasible for him to always read the full victim memory because of the data transfer rate being the frequency limiting factor. Once the address space is identified, the next step is to identify the two memory regions allocated to each of the local variables R_0 and R_1 (cf. Algorithm 7) within the victim application's address space. This allows any observed change in either of these two regions to be linked with an update to the variables R_0 and R_1 respectively. Finally, the updates in R_0 and R_1 memory regions are observed via frequent snapshots for a period of one complete encryption, and the order of these updates is linked back to Algorithm 7 to learn the key k. We explain these steps in detail in the following subsections.

5.2.3 Step 1: Application's Address Space Identification

Since the application is supposed to be continuously updating its data (e.g., variables R_0, R_1), its address space can be identified by finding the memory regions which are continuously being updated. Algorithm 8

Algorithm 9 Pseudo code for the second phase of attack

Input: S: Application's memory space. (from Algorithm 8);
n: # of snapshots to cover one full encryption period.
Output: k: Application's secret key.
Start:

1: $V = (s_1, \cdots, s_n) \mid s_i = \text{TAKESNAPSHOT}(S), 1 \leq i \leq n$
2: $Th = \text{COMPUTETHRESHOLD}(V)$
3: $W = \emptyset$, $k = (0, \cdots, 0)$
4: $V = \text{REMOVEUNCHANGEDSNAPSHOTS}(V)$
5: **for** $i = 1$ **to** $|V| - 1$ **do**
6: $\quad R_{x_i} = \text{CORRELATE}(s_i, s_{i+1}, Th)$ ▷ $x_i \in \{0, 1\}$
7: $\quad W = W \cup R_{x_i}$
8: **end for**
9: $i = 1$, $j = 0$
10: **for** $(R_{x_i}, R_{x_{i+1}}) \in W$ **do**
11: $\quad$ **if** $R_{x_i} = R_0$ and $R_{x_{i+1}} = R_1$ **then** $k_j = 1$
12: $\quad$ **else if** $R_{x_i} = R_1$ and $R_{x_{i+1}} = R_0$ **then** $k_j = 0$
13: $\quad$ **end if**
14: $\quad i = i + 2; \quad j = j + 1$
15: **end for**
return k

shows this process at an abstract level. The whole of the victim system's memory space is divided into M blocks, each of some reasonable size B (say a few megabytes). Two subsequent snapshots of each block $m \in M$ are compared with each other through COMPAREMATCH procedure. It is a heuristic based process which searches for a specific pattern of updates between the two snapshots which potentially represents the application's footprint. For example, a sequence of two modified consecutive 64 byte cache lines followed by a few unmodified cache lines and then further two modified consecutive cache lines would potentially represent the two 128 byte regions for R_0 (first two cache lines) and R_1 (last two cache lines). Finally, a set S of all those memory blocks which show the specific update sequence searched by COMPAREMATCH is returned. This algorithm is iteratively repeated until a reasonably small set of memory block(s) (e.g., one 4 kB page) is identified which is expected to contain the victim application's address space.

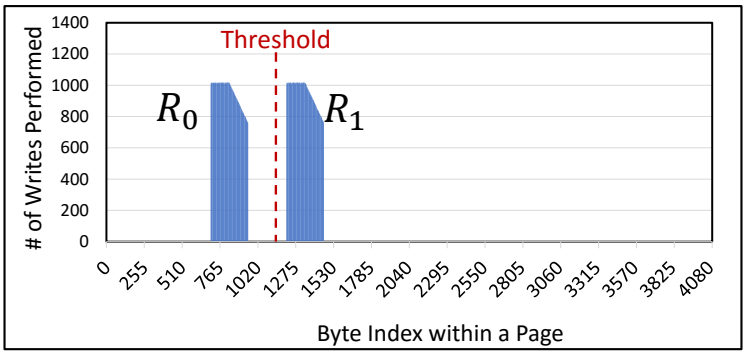


Figure 5.2.2: A histogram of # of writes to individual bytes in the victim's memory page. A clear distinction is shown between the regions corresponding to variables R_0 and R_1.

5.2.4 Step 2: Distinguishing Local Variables R_0 and R_1

Once the application's memory space is found, we need to link two distinct regions within this address space to the variables R_0 and R_1 in order to determine the key bits from the order of their updates. For this purpose, a set V of n snapshots of the application's space is computed as shown in Algorithm 9. Notice that n is large enough to cover one full encryption period. The COMPUTETHRESHOLD procedure computes a histogram of the updates performed inside the application's memory over all the snapshots of set V. Figure 5.2.2 shows one such histogram for a 4kB page of victim's memory. It can be seen that almost all the updates are performed at two distinct regions spanning over only a few cache lines within the page. These two regions correspond to the variables R_0 and R_1 respectively[2]. The *inactive* region between R_0 and R_1 represents a threshold which is later used by CORRELATE procedure to determine whether a change in two successive memory snapshots corresponds to an update in R_0 or R_1 etc.

5.2.5 Step 3: Inferring the Secret Key

After computing the set of snapshots V and the threshold Th, we enter the final phase of inferring the secret key (starting from step 4 in Algorithm 9). Up to this point, the sequence V contains pairs of snapshots that represent changes in R_0 and R_1, and also the pairs which represent no change, as shown in Figure 5.2.3. The

[2]We can tell whether R_0 or R_1 comes first in the memory layout from the declaration order of these variables in the actual implementation of the exponentiation algorithm (cf. line 1 in Algorithm 7).

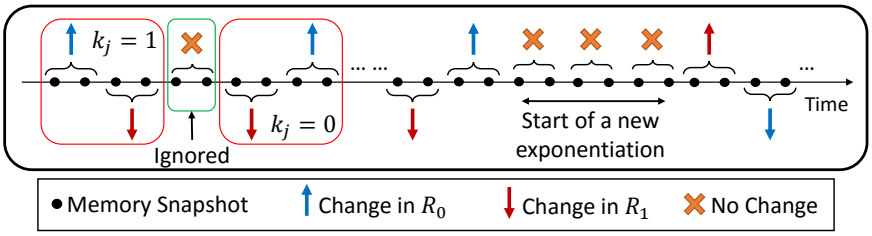


Figure 5.2.3: Inferring the secret key via observing the sequence of snapshots and the changes in variables R_0 and R_1. The pairs of snapshots which do not show any change are ignored.

reason why some pairs do not show any change is because our snapshot frequency is higher than the rate at which the application updates its data. This allows us to learn the write access pattern at a fine granularity.

In order to learn the write access pattern, first the pairs of unchanged snapshots from the sequence V are removed by the procedure REMOVEUNCHANGEDSNAPSHOTS(V). The resulting sequence V only contains pairs which always represent a change, either in R_0 or R_1. Now, each pair of two successive snapshots is correlated to an update in either R_0 or R_1 by CORRELATE procedure using the threshold computed earlier.

As mentioned earlier, Montgomery ladder algorithm performs computations upon local variables, where the order of variable updates is based on the secret exponent bits (cf. Algorithm 7). Therefore, judging from the order of updates made in R_0 and R_1, each pair of updates $(R_{x_i}, R_{x_{i+1}}) \in W$ is linked back to the corresponding value of the secret key bit k_j as shown in figure 5.2.3. As the set W contains the history of all the updates to R_0 and R_1 for a complete encryption, therefore all the key bits can be inferred through the above mentioned process.

5.3 Attack Demonstration

5.3.1 Experimental Setup

Our experiment setup uses two computer systems, one being the attacker and the other being the victim. In our experiments, the victim system is *DELL XPS 8700*, comprising of *Intel Core(TM) i7-4790 3.60GHz* processor that uses *Ubuntu 14.04.3 LTS* operating system with a *Linux kernel 3.19.0-43-generic*, and has

16GB of main memory. The attacker machine is a *64-bit Windows 10* based system having 8GB of main memory. A PCI adapter module, called *USB Evaluation Board* [88], is connected to the victim via the PCI-Express slot and acts as a compromised DMA device (cf. Figure 5.2.1). This DMA device, together with PCILeech software [85], allows the attacker to monitor victim's memory and/or take its snapshots. To implement our attack, the PCILeech software has been extended to first find the application's address space in the victim's memory (cf. Algorithm 8), and then attack the identified address space to infer the secret key (cf. Algorithm 9). The above mentioned attacking algorithms run *while* the victim application is executing. The victim system has a BIOS version *A11* which supports *write-through* enabled L1 and L2 caches while disabling the L3 cache by default. Hence, any updates made by the application are immediately propagated to the main memory as each multiplication/squaring write operation in performed. Section 5.3.2 explains the step by step details about how the attack is launched.

5.3.2 Experimental Results

In order to take memory snapshots via PCI module and the PCILeech software, the attacker first needs to load a kernel module into the victim system via the PCI module itself. Notice that the attacker does not require any extra privileges to do so. We use the following command via PCILeech software to load the kernel into victim's DRAM. When the kernel is loaded, an address is spitted out by the software, which shows where the module resides in the victim's memory. Loading the kernel into memory is a rapid process and takes only a few milliseconds to complete the process.

```
D:\>pcileech kmdload -kmd LINUX_X64
KMD: Code inserted into the kernel
KMD: Execution received - continuing ...
KMD: Successfully loaded at 0x1b54a000
D:\>_
```

In the meantime, the Montgomery's ladder exponentiation algorithm is run on the victim machine using a 128 byte (1024 bits) message along with a secret key of 64 bytes (512 bits).

```
[user@victim]$ ./montgomery_exponentiation
```

With the application running and the kernel module loaded into victim's memory, we proceed to find the potential regions in the DRAM which are being accessed frequently by taking multiple snapshots. To retrieve these snapshots, we issue the *pagefind* command shown below which uses the loaded kernel module's address to access the victim's full memory.

We integrated the *pagefind* command into the PCILeech software to iteratively find regions getting modified persistently. *pagefind* narrows down the selected regions to a single page by constantly monitoring and comparing the changes being made, and returns the address of the page where application's array data structures are defined. This step corresponds to *Application's Address Space Identification* phase of the attack (cf. Algorithm 8) and is the most time consuming phase. To read the whole memory, comparing their respective snapshots and narrowing down to a single page of 4KB from 16GB search space takes ∼3 minutes and 30 seconds.

```
D:\>pcileech pagefind -kmd 0x1b542000
Matching Pattern ...
Page Finding:  Successful.
Total_Time = 210199 Milliseconds
Victim Page Address :  0xd271c000
D:\>_
```

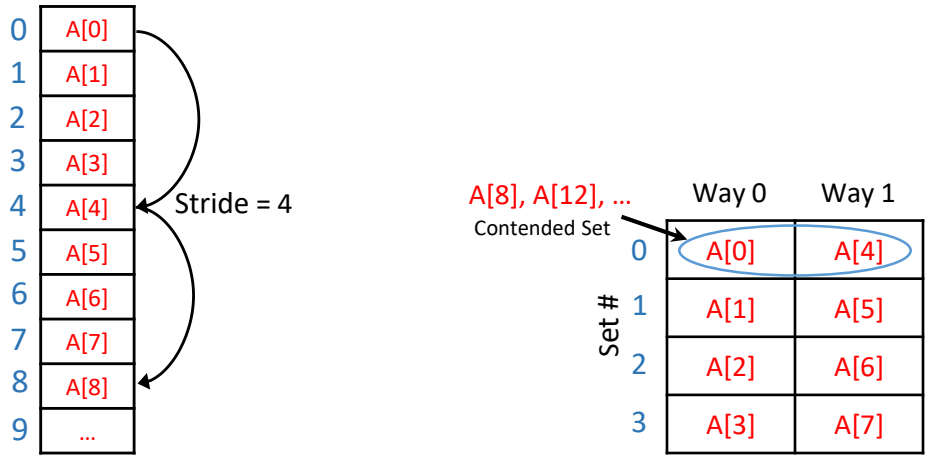
As shown above, from the first phase we retrieve the address of the page where application's data structures are stored. Proceeding towards our second and third step namely *Distinguishing Local Variables* and *Inferring the Secret Key* (cf. Section 5.2.4, 5.2.5), we use another integrated command *pageattack*. It first takes a predefined number of snapshots of the application page provided by the first step, and distinguishes the message (R_1) and algorithm result (R_0) from the rest of the stale data, residing on the memory page. It then uses the order of changes in R_0 and R_1 to infer the secret key.

```
D:\>pcileech pageattack —min 0xd271c000
Attack Successful.
Total_Time = 3596 Milliseconds
Inferred Key is:
1a 4b 28 41 e6 27 d4 7d
72 c3 40 79 be 1f 6c 35
ca 3b 58 b1 96 17 04 ed
22 b3 70 e9 6e 0f 9c a5
7a 2b 88 21 46 07 34 5d
d2 a3 a0 59 1e ff cc 15
2a 1b b8 91 f6 f7 64 cd
82 93 d0 c9 ce ef fc 85
D:\>_
```

This final step takes ∼3.6 seconds to complete and returns the complete 512 bit secret key learned from only the write access patterns. Combing the times associated with all the attack phases, the total attack time comes out to be ∼3 minutes, 34 seconds.

5.4 Leakage under Caching Effects

In view of our proposed attack on Montgomery ladder based exponentiation algorithm, the updates to the application data should always be available in the DRAM of the victim system before an attacker issues a memory snapshot request. This is only possible if the victim system has *write-through* enabled cache hierarchy or the caches are disabled altogether. Whereas, on the other hand, modern processors typically consist of large on-chip *write-back* caches where the updates to application's data are only be visible in DRAM once the data is evicted from the last level cache (LLC). Thus in the attack proposed in Section 5.2, the caching effects are not catered for, which introduce 'noise' to the precise write-access sequence inferred earlier, hence making the attacker's job difficult. A possible workaround to deal with such caching effects is to collect several 'noisy' sequences of memory snapshots and then run correlation analysis on them to learn



(a) Accessing an array with a stride of 4.

(b) A 2-way set-associative cache.

Figure 5.4.1: A strided memory access pattern causing contention on a single cache set.

the precise write-access pattern. Furthermore, if the adversary is also a user of the same computer system, it can flush the system caches frequently to reduce the noise in write-access sequence even further.

Another (more efficient) attack scenario under write-back caches would be when the application has a strided memory access pattern that causes contention over the cache sets, and hence forces its own data to be evicted to make room for the new data in the cache. In the following subsection, we discuss how such a strided memory access pattern can lead to evictions to the DRAM which could potentially leak private information.

5.4.1 Memory Striding and Cache Set Contention

A strided memory access pattern is the one where each request to the memory is for the same number of bytes, and the access pointer is incremented by the same amount between each request. An array accessed with a stride of exactly the same size as the size of each of its elements results in accessing contiguous locations in the memory. Such access patterns are said to have a stride value of 1. Figure 5.4.1a shows a *non-unit* striding access pattern in which the elements $0, 4, 8, 12, \cdots$ of an array A are accessed. This access pattern has a stride value of 4.

Consider a simple system which has a 2-way set-associative *write-back* cache with a total capacity of 8 cache lines, as shown in Figure 5.4.1b. The strided access pattern from Figure 5.4.1a accesses every $4i_{th}$ element of the array A, where $i = 0, 1, 2, \cdots$. Assuming that each element of A is of size equal to the cache line size, for a simple *modulus based* cache hash function, the elements $A[0], A[4], A[8], \cdots$ are mapped to the same set causing contention over *set* 0. Since the cache associativity is only 2, this access sequence causes evictions from the cache when both ways of set 0 contain valid cache lines. Similarly, elements $A[1], A[5], A[9], \cdots$ map to *set* 1, and this access sequence will cause evictions from set 1, and so on. In other words, such write-access sequences are still propagated almost immediately to the next level in the memory hierarchy (e.g., DRAM) even under write-back caches, which could potentially leak information. This is an artifact of the cache implementation combined with the striding access pattern of the application.

It must be noted that, not all evictions result in updates to the main memory. Typically, only *dirty* cache lines, caused by data writes, evicted from the cache are propagated to the main memory. *Clean* evictions from the cache are simply discarded resulting in no change in the main memory since it already contains a clean copy of the data.

Assume that an application generates two distinguishable striding write-access patterns that result in contention at two different cache sets, leading to evictions from the cache. Consequently, the resulting write-access access sequence will be revealed to an adversary who is capable of monitoring changes in the main memory, potentially resulting in privacy leakage.

5.4.2 Striding Application: Gaussian Elimination

In Section 5.4.1 we discussed, using a toy example, how a strided access pattern can lead to information leakage. Now we present a realistic example which has such a striding access pattern, and later in Section 5.4.3 we show how such a pattern can be exploited to learn private information. We consider the application of *Gaussian Elimination* of large binary matrices carrying substantial amount of information. Clearly, these large matrices cannot fit into the caches, therefore there will be cache evictions as a result of Gaussian elimination operations.

Gaussian elimination a.k.a. row reduction is a method for solving system of linear equations by the use

of matrices in the form $Ax = B$. Row reduction is done by doing a series of elementary row operations which modify the matrix until it forms an upper triangular matrix, i.e., elements underneath the main diagonal are zeros. Different types of elementary row operations include swapping two rows, multiplying a row by a non-zero number and adding a multiple of one row to another. The upper triangular matrix formed out of these operations will be in row echelon form. When the leading coefficient (pivot) in each row is 1, and every column containing the leading coefficient has zeros elsewhere, the matrix is said to be in reduced row echelon form. The Gaussian elimination algorithm consists of two processes, one being *forward elimination* that converts the matrix to row echelon form and the other is *backward substitution* that calculates values of the unknowns. These processes result in solving the linear equation.

Gauss-Jordan elimination uses a similar approach for finding the inverse of a matrix. For a $n \times n$ square matrix S, elementary row operations can be applied to reduce the matrix into reduced echelon form, and furthermore, for computing the matrix inverse if it exits. Initially, the $n \times n$ identity matrix I is augmented to the right of S, forming a $n \times 2n$ block matrix $[S|I]$. Now, upon applying the row operations, the left block can be reduced to the identity matrix I if S is invertible. This gives S^{-1} which is the right block of the final matrix. In a nutshell, we continue performing row operations until $[S|I]$ becomes $[I|S^{-1}]$.

Consider that the matrix under elimination is stored in a *column-contiguous* manner in the computer system's main memory. In other words, each column occupies a contiguous chunk of memory equal to the column size, after which the next column resides, and so on. When consecutive elements of a row of this matrix are accessed during a row operation, the corresponding memory access pattern results in a striding sequence, where the stride length is equal to the column size. If the stride length is such that it creates contention on particular cache sets corresponding to particular rows, this would reveal the modified row, which in turn could potentially leak the binary matrix itself (cf. Section 5.4.3).

5.4.3 Attacking McEliece Public-Key Cryptosystem

McEliece public key cryptosystem [89], an asymmetric encryption algorithm, uses an error correcting code for a description of the private key. This encryption uses a fast and efficient decoding algorithm, namely a Goppa code and hides the structure of the code by transformation of the generator matrix. This transforma-

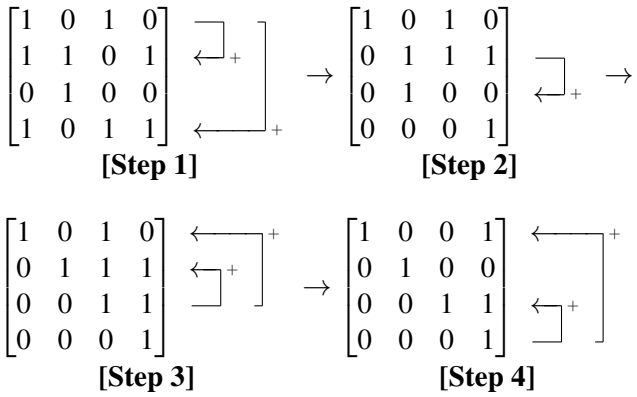


Figure 5.4.2: The Gaussian Elimination process on a 4×4 binary matrix.

tion yields the public key and the structure of the Goppa code together with the transformation parameters, which further provides the trapdoor information. For a linear code C, generator matrix G, random invertible matrix S and random permutation matrix P, the matrix $G^* = SGP$ is made public while P, G, and S form the private key. A message m is encrypted along with a random error vector using the equation $c = mG^* + e$, where c refers to the ciphertext. In the decryption process, we compute $c^* = cP^{-1}$, decode c^* to m^* by the decoding algorithm, and lastly compute $m = m^*S^{-1}$. Notice, that S is a private binary matrix whose inverse is used to recover the message m. Any system carrying out this encryption/decryption process could either store the matrix inverse (for better performance) or calculate the inverse during the run time. However, the latter could lead to the leakage of the binary matrix via write-access patterns during the inverse computation. In this section we will demonstrate how performing Gauss-Jordan elimination [90] on the binary secret matrix S could lead to its complete exposure as a consequence of cache striding and cache set contention as shown in section 5.4.1.

For the ease of demonstration we consider a 4×4 binary matrix. The elements stored in the main memory are column contiguous. We assume that each element of the matrix is *cache line aligned* for performance reasons. In other words, each element is stored in a unique cache line in order to avoid false sharing within a cache line. Considering a system with a 2-way 4-set associative cache, each row of the matrix is mapped to one cache set due to the cache structure (cf. Figure 5.4.1b). The elimination process to

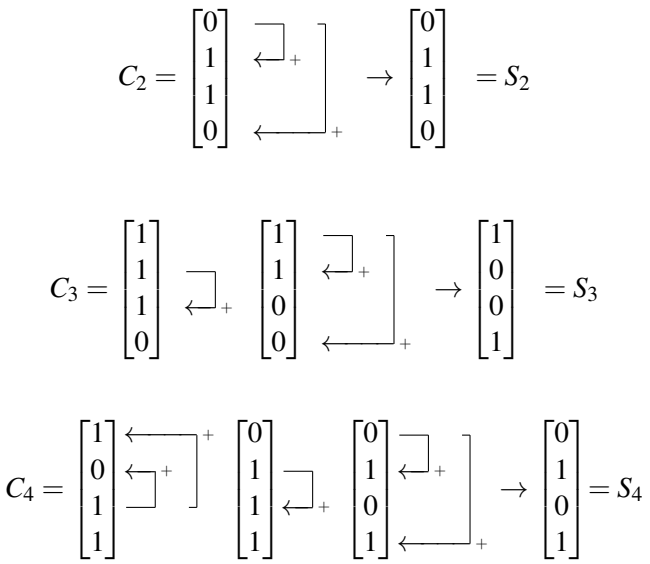


Figure 5.4.3: Back Substitution process to recover secret binary matrix S.

obtain the inverse of the binary matrix $S_{4\times4}$ is shown step by step in Figure 5.4.2. After these row operations, we obtain an identity matrix $I_{4\times4}$.

In each of the above 4 steps, the corresponding pivot row is added to another row or rows. For example, in **Step [1]** row 1 is added to row 2 and row 4. Similarly, row 2 is added to row 3 in **Step [2]**. As a row operation is performed, the elements of the target row are modified and result in cache line evictions, since accessing a whole row causes contention over the corresponding set it is mapped to, and a cache set can only store 2 elements of a row at a time. For instance, in **Step [1]**, row 2 and row 4 are modified causing contention and evictions from set 1 and 3 respectively. Consequently, an adversary can learn the identifier of the row being updated during each row operation by monitoring the address space in which any updates take place, and then linking it back to the row number. Now, by definition of the elimination algorithm, all column elements corresponding to rows that undergo addition operations can be inferred as 1s, and the remaining ones as 0s. Hence, in each of **Step [1]**, **Step [2]**, **Step [3]** and **Step [4]**, we infer the corresponding pivot column to be $C_1 = \{1,1,0,1\}$, $C_2 = \{0,1,1,0\}$, $C_3 = \{1,1,1,0\}$ and $C_4 = \{1,0,1,1\}$ respectively.

Notice that C_2, C_3, and C_4 obtained in the above steps show the respective intermediate forms of the

corresponding columns of S during the elimination process. These values, however, can be used to recover the original column values of matrix S through *back substitution* process, as shown in Figure 5.4.3. In this process, each column C_i undergoes the row operations performed (and inferred) in each of the steps **Step [i-1]**, **Step [i-2]**, $\cdots$, **Step [1]**, precisely in this order. For example, C_2 undergoes addition of row 1 to rows 2 and 4, while C_3 performs addition of row 2 to row 3 along with the addition of row 1 to rows 2 and 4. Upon completion of the back substitution process, the complete secret matrix S is recovered by the adversary.

5.5 Chapter Review

Privacy leakage via purely write-access patterns is less obvious and not extensively studied in the current literature. This chapter demonstrates a real attack on Montgomery's ladder based modular exponentiation algorithm and infer the secret exponent by just learning the write access patterns of the algorithm to the main memory. It adapts the traditional DMA based exploits to learn the application's write access pattern in a reasonable time. The proposed attack takes just 3 minutes and 34 seconds to learn 512 secret bits from a typical Linux based victim system. A possible attack on McEliece public-key cryptosystem has also been presented.

Chapter 6

Flat ORAM: A Simplified Write-Only Oblivious RAM

A key observation regarding the adversarial model assumed by the current renowned ORAM techniques is that the adversary is capable of learning fine-grained information of *all* the accesses made to the memory. This includes the information about which location is accessed, the type of operations (read/write), and the time of the access. It is an extremely strong adversarial model which, in most cases, requires direct physical access to the memory address bus in order to monitor both read *and* write accesses, e.g. the case of a *curious* cloud server.

On the other hand, for purely remote adversaries (where the cloud server itself is trusted), direct physical access to the memory address bus is not possible thereby preventing them from directly monitoring read-/write access patterns. Such remote adversaries, although weaker than adversaries having physical access, can still "learn" the application's write access patterns. Interestingly, privacy leakage is still possible even if the adversary is able to infer just the write access patterns of an application.

John *et al.* demonstrated such an attack [28] on the famous Montgomery's ladder technique [29] commonly used for modular exponentiation in public key cryptography. In this attack, a 512-bit secret key is correctly inferred in just 3.5 minutes by only monitoring the application's write access pattern via a compro-

mised Direct Memory Access (DMA[1]) [91, 84, 92, 93, 94] device on the system. The adversary collects the *snapshots* of the application's memory via the compromised DMA. Clearly, any two memory snapshots only differ in the locations where the data has been modified in the latter snapshot. In other words, comparing the snapshots not only reveals the fact that write accesses (if any) have been made to the memory, but it also reveals the exact locations of the accesses which leads to a precise access pattern of memory writes resulting in privacy leakage.

Recent work [87] demonstrated that DMA attacks can also be launched *remotely* by injecting malware to the dedicated hardware devices, such as graphic processors and network interface cards, attached to the host platform. This allows even a remote adversary to learn the application's write access pattern. Intel's TXT has also been a victim of DMA based attacks where a malicious OS directed a network card to access data in the protected VM [95, 96].

One approach to prevent such attacks, as adapted by TXT, could be to block certain DMA accesses through modifications in DRAM controller. However, this requires the DRAM controller to be included in the trusted computing base (TCB) of the system which is undesirable. Nevertheless, there could potentially be many scenarios other than DMA based attacks where write access patterns can be learned by the adversary.

Current so-called *fully functional* ORAM schemes, which obfuscate both read *and* write access patterns, also offer a solution to such weaker adversaries. However, the added protection (obfuscation of reads) offered by fully functional ORAMs is unnecessary and is practically an overkill in this scenario which results in significant performance penalty. Path ORAM [19], the most efficient and practical fully functional ORAM system for secure processors so far, still incurs about $2-10\times$ performance degradation [10, 20] over an insecure DRAM system. A far more efficient solution to this problem is a *write-only* ORAM scheme, which only obfuscates the write accesses made to the memory. Since read accesses leave no trace in the memory snapshot and hence do not need to be obfuscated in this model, a write-only ORAM can offer significant performance advantage over a fully functional ORAM.

[1] DMA is a standard performance feature which grants full access of the main memory to certain peripheral buses, e.g. FireWire, Thunderbolt etc.

A recent work, HIVE [30], has proposed a write-only ORAM scheme for implementing hidden volumes in hard disk drives. The key idea is similar to Path ORAM, i.e., each data block is written to a new random location, along with some dummy blocks, every time it is accessed. However, a fundamental challenge that arises in this scheme is to avoid *collisions*. I.e., to determine whether a randomly chosen physical location contains *real* or *dummy* data, in order to avoid overwriting the existing useful data block. HIVE proposes a complex *inverse position map* approach for collision avoidance. Essentially, it maintains a forward position mapping (logical to physical blocks) and a backward/inverse position mapping (physical to logical blocks) for the whole memory. Before each write, both forward and backward mappings are looked up to determine whether the corresponding physical location is vacant or occupied. This approach, however, turns out to be a storage and performance bottleneck because of the size of inverse position map and dual lookups of the position mappings.

A simplistic and obvious solution to this long-standing problem, from a computer architect's perspective, would be to use a bit-mask to mark each physical block as vacant or occupied. Based on this intuition, a simplified write-only ORAM scheme called Flat ORAM[2] is proposed. At the core of the algorithm, Flat ORAM introduces a new data structure called Occupancy Map (OccMap): a bit-mask which offers an efficient collision avoidance mechanism. The OccMap records the availability information (occupied/vacant) of every physical location in a highly compressed form (i.e., just 1-bit per cache line). For typical parameter settings, OccMap is about 25× compact compared to HIVE's inverse position map structure. This dense structure allows the OccMap blocks to exploit higher locality, resulting in considerable performance gain. While naively storing the OccMap breaks the ORAM's security, we present how to securely store and manage this structure in a real system. In particular, this chapter makes the following contributions:

1. This work is the first one to implement an existing write-only ORAM scheme, HIVE, in the context of secure processors with all state-of-the-art ORAM optimizations, and analyze its performance bottlenecks.

2. A simple write-only ORAM, named Flat ORAM, having an efficient collision avoidance approach is

[2]Flat ORAM replaces the binary tree structure of Path ORAM with a *flat* array of data blocks; hence termed as 'Flat' ORAM.

proposed. The micro-architecture of the scheme is discussed in detail and the design space is comprehensively explored. It has also been shown to seamlessly adopt various performance optimizations of its predecessor: Path ORAM.

3. The simulation results show that, on average, Flat ORAM offers 50% performance gain (up to 75% for DBMS) over HIVE, and only incurs slowdowns of 3× and 1.6× over the insecure DRAM for memory bound Splash2 and SPEC06 benchmarks respectively.

The rest of this chapter is organized as follows: Section 6.1 describes our adversarial model in detail along with a practical example from the current literature. Section 6.2 provides the necessary background of fully functional ORAMs and write-only ORAMs. The proposed Flat ORAM scheme along with its security analysis is presented in Section 6.3, and the detailed construction of its occupancy map structure is shown in Section 6.4. A few additional optimizations from literature implemented in Flat ORAM are discussed in Section 6.5. Section 6.6 evaluates Flat ORAM's performance.

6.1 Adversarial Model

We assume a relaxed form of the adversarial model considered in several prior works related to fully functional oblivious RAMs in secure processor settings [20, 48, 61].

In our model, a user's computation job is outsourced to a cloud, where a trusted processor performs the computation on user's private data. The user's private data is stored (in encrypted form) in the untrusted memory external to the trusted processor, i.e. DRAM. In order to compute on user's private data, the trusted processor interacts with DRAM. The cloud service itself is not considered as an adversary, i.e. it does not try to infer any information from the memory access patterns of the user's program. However, since the cloud serves several users at the same time, sharing of critical resources such as DRAM among various applications from different users is inevitable. Among these users being served by the cloud service, we assume a malicious user who is able to monitor remotely (and potentially learn the secret information from) the data write sequences of other users' applications to the DRAM, e.g., by taking frequent snapshots of

the victim application's memory via a compromised DMA. Moreover, he may also tamper with the DRAM contents or play replay attacks in order to manipulate other users' applications and/or learn more about their secret data.

To protect the system from such an adversary, we add to the processor chip a *Write-Only ORAM controller*: an additional trusted hardware module. Now all the off-chip traffic goes to DRAM through the ORAM controller. In order to formally define the security properties satisfied by our ORAM controller, we adapt the write-only ORAM *privacy* definition from [97] as follows:

Definition 6.1.1. (Write-Only ORAM Privacy) For every two logical access sequences A_1 and A_2 of infinite length, their corresponding (infinite length) probabilistic access sequences $\mathsf{ORAM}(A_1)$ and $\mathsf{ORAM}(A_2)$ are identically distributed in the following sense: For all positive integers n, if we truncate $\mathsf{ORAM}(A_1)$ and $\mathsf{ORAM}(A_2)$ to their first n accesses, then the truncations $[\mathsf{ORAM}(A_1)]_n$ and $[\mathsf{ORAM}(A_2)]_n$ are identically distributed.

In other words, memory snapshots only reveal to the adversary the *timing* of write accesses made to the memory (i.e. leakage over *ORAM Timing Channel*) instead of their precise access pattern, whereas no trace of any *read* accesses made to the memory is revealed to the adversary. An important aspect of Definition 6.1.1 to note is that it completely isolates the problem of leakage over *ORAM Termination Channel* from ORAM's originally targeted problem (which is also targeted in this work) i.e., preventing leakage over memory address channel. Notice that the original definition of ORAM [18] does not protect against timing attacks, i.e. it does not obfuscate *when* an access is made to the memory (ORAM Timing Channel) or how long it takes for the application to finish (ORAM Termination Channel). The write-only ORAM security definition followed by HIVE [30] also allows leakage over ORAM termination channel, as two memory access sequences generated by the ORAM for two same-length logical access sequences can have different lengths [97]. Therefore, in order to define precise security guarantees offered by our ORAM, we follow Definition 6.1.1.

Periodic ORAMs [10] deterministically make ORAM accesses always at regular predefined (publicly known) intervals, thereby preventing leakage over ORAM timing channel and shifting it to the ORAM

Algorithm 10 Montgomery Ladder

Inputs: $g, k = (k_{t-1}, \cdots, k_0)_2$ **Output:** $y = g^k$
Start:

$R_0 \leftarrow 1; R_1 \leftarrow g$
for $j = t-1, 0$ **do**
  if $k_j = 0$ **then** $R_1 \leftarrow R_0 R_1$; $R_0 \leftarrow (R_0)^2$
  else $R_0 \leftarrow R_0 R_1$; $R_1 \leftarrow (R_1)^2$
  end if
end for

return R_0

termination channel. We present a periodic variant of our write-only ORAM to protect leakage over ORAM timing channel. Following the prior works [11] [61], (a) we assume that the timing of individual DRAM accesses made *during* an ORAM access does not reveal any information; (b) we do not protect the leakage over ORAM termination channel (i.e. total number of ORAM or DRAM accesses). The problem of leakage over ORAM termination channel has been addressed in the existing literature [62] where only $\log_2(n)$ bits are leaked for a total of n accesses. A similar approach can easily be applied to the current scheme.

In order to detect malicious tampering of the memory by the adversary, we follow the standard definition of data *integrity* and *freshness* [61]:

Definition 6.1.2. (Write-Only ORAM Integrity) From the processor's perspective, the ORAM behaves like a valid memory with overwhelming probability, and detects any violation to data authenticity and/or freshness.

6.1.1 Practicality of the Adversarial Model

Modular exponentiation algorithms, such as RSA algorithm, are widely used in public-key cryptography. In general, these algorithms perform computations of the form $y = g^k \mod n$, where the attacker's goal is to find the secret k. Algorithm 10 shows the Montgomery Ladder scheme [29] which performs exponentiation (g^k) through simple square-and-multiply operations. For a given input g and a secret key k, the algorithm performs multiplication and squaring operations on two local variables R_0 and R_1 for each bit of k starting

from the most significant bit down to the least significant bit. This algorithm prevents leakage over power side-channel since, regardless of the value of bit k_j, the same number of operations are performed in the same order, hence producing the same power footprint for $k_j = 0$ and $k_j = 1$.

Notice, however, that the specific order in which R_0 and R_1 are updated in time depends upon the value of k_j. E.g., for $k_j = 0$, R_1 is written first and then R_0 is updated; whereas for $k_j = 1$ the updates are done in the reverse order. This sequence of write access to R_0 and R_1 reveals to the adversary the exact bit values of the secret key k. A recent work [28] demonstrated such an attack where frequent memory snapshots of victim application's data (particularly R_0 and R_1) from the physical memory are taken via a compromised DMA. These snapshots are then correlated in time to determine the sequence of write access to R_0, R_1, which in turn reveals the secret key. The reported time taken by the attack is 3.5 minutes.

One might argue that under write-back caches, the updates to application's data will only be visible in DRAM once the data is evicted from the LLC. This will definitely introduce noise to the precise write-access sequence discussed earlier, hence making the attacker's job difficult. However, he can still collect several 'noisy' sequences of memory snapshots and then run correlation analysis on them to find the secret key k. Furthermore, if the adversary is also a user of the same computer system, he can flush the system caches frequently to reduce the noise in write-access sequence even further.

6.2 Background of Oblivious RAMs

A fully functional Oblivious RAM [18], or more commonly known as ORAM, is a primitive that obfuscates the user's (i.e. Processor's) access patterns to a storage (i.e. DRAM) such that by monitoring the memory access patterns, an adversary is not able to learn anything about the data being accessed. The ORAM interface transforms the user's access sequence of program addresses into a sequence of ORAM accesses to random looking physical addresses. Since the physical locations being accessed are revealed to the adversary, the ORAM interface guarantees that the physical access pattern is independent of the logical access pattern hence user's potentially data dependent access patterns are not revealed. Furthermore, the data stored in ORAMs should be encrypted using probabilistic encryption to conceal the data content as

well as the fact whether or not the content has been updated.

6.2.1 Path ORAM

Path ORAM [19] is currently the most efficient and well studied ORAM implementation for secure processors. It has two main hardware components: the *binary tree storage* and the *ORAM controller*. Binary tree stores the data content of the ORAM and is implemented on DRAM. Each node in the tree can hold up to Z useful data blocks, and any empty slots are filled with dummy blocks. All blocks, real or dummy, are probabilistically encrypted and cannot be distinguished. The path from the root node to the leaf s is defined as path s. ORAM controller is a piece of trusted hardware that controls the tree structure. Besides necessary logic circuits, the ORAM controller contains two main structures, a *position map* and a *stash*. The *position map* is a lookup table that associates the program address a of a data block with a path in the ORAM tree (path s). The *stash* is a buffer that stores up to a small number of data blocks at a time.

Each data block a in Path ORAM is mapped (randomly) to some path s via the position map, i.e. at any time, the data block a must be stored either on path s, or in the stash. Path ORAM follows the following steps when a request on block a is issued by the processor: (1) The path (leaf) number s of the logical address a is looked up in the position map. (2) All the blocks on path s are read and decrypted, and all real blocks added to the stash. (3) Block a is returned to the processor. (4) The position map of a is updated to a new random leaf s'. (5) As many blocks from stash as possible are encrypted and written on path s, where empty spots are filled with dummy blocks. Step (4) guarantees that when block a is accessed later, a random path will be accessed which is independent of any previously accessed paths (*unlinkability*). As a result, each ORAM access is random and unlinkable regardless of the request pattern.

Path ORAM incurs significant energy and performance penalties compared to insecure DRAM. Under typical settings for secure processors (gigabytes of memory and 64- to 128-byte blocks), Path ORAM has a 20-30 level binary tree where each node typically stores 3 or 4 data blocks [57, 20]. This means that each ORAM access reads *and* writes 60-120 blocks, in contrast to a single read *or* write operation in an insecure storage system.

6.2.2 Write-Only ORAMs

In contrast to fully functional ORAMs, a write-only ORAM only obfuscates the patterns of *write* accesses made to a storage. Write-only ORAM is preferred for performance reasons over fully functional ORAMs in situations where the adversary is not able to monitor the *read* access patterns.

There has been very limited research work done so far to explore write-only ORAMs, and to best of our knowledge, write-only ORAMs for secure processors have not been explored at all. Li and Datta [98] present a write-only ORAM scheme to be used with Private Information Retrieval (PIR) in order to preserve the privacy of data outsourced to a data center. Although this scheme achieves an amortized write cost of $O(B \log N)$, it incurs a read cost of $O(B.N)$ for a storage of N blocks each of size B. For efficient reads, it requires the client side storage (i.e. the on-chip position map) to be polynomial in N. In a secure processor setting, DRAM reads are usually the major performance bottleneck, and introducing a complexity polynomial in N on this critical path is highly unwanted.

A recent work, HIVE [30], has proposed a write-only ORAM scheme for hidden volumes in hard disk drives. Although HIVE write-only ORAM presented-as-is [30] targets a totally different application, we believe that its parameter settings can be tweaked to be used in the secure processor setting. This work is the first one to implement HIVE in the secure processor context, and we consider this implementation as the baseline write-only ORAM to be compared with our proposed Flat ORAM.

Roche *et al.* [99] have also proposed an efficient write-only ORAM scheme for hard disk storages. This scheme along with its complex optimizations has been implemented in software which is completely feasible at the DRAM-Disk boundary. However, in this work, we target the Processor-DRAM boundary for our proposed ORAM which needs to be implemented in hardware, and hence our focus is only towards simplified algorithms and optimizations which can easily be synthesized in hardware without substantial area overhead.

6.3 Flat ORAM Scheme

In this section, we first present the core algorithm of Flat ORAM, then we discuss its architectural details and various optimizations for a practical implementation.

6.3.1 Fundamental Data Structures

Position Map (PosMap): It is a standard Path ORAM structure that maintains randomized mappings of logical blocks to physical locations. However there is one subtle difference between PosMap of Path ORAM and Flat ORAM. In Path ORAM, PosMap stores a path number for each logical block and the block can reside *anywhere* on that path. In contrast, PosMap in Flat ORAM stores the exact physical address where a logical block is stored.

Occupancy Map (OccMap): OccMap is a newly introduced structure in Flat ORAM. It is essentially a large bit-mask where each bit corresponds to a physical location (i.e., a cache line). The binary value of each bit represents whether the corresponding physical block contains real or outdated/dummy data. OccMap is of crucial importance to avoid *data collisions*, and hence for the correctness of Flat ORAM. A collision happens when a physical location, which is randomly chosen to store a logical block, already contains useful data which cannot be overwritten. Managing the OccMap securely and efficiently is a major challenge which we address in section 6.4 in detail.

Stash: Stash, also adapted from Path ORAM, is a small buffer in the trusted memory to temporarily hold data blocks evicted from the processor's last level cache (LLC). A slight but crucial modification, however, is that Flat ORAM only buffers *dirty*[3] data blocks in the stash; while *clean* blocks evicted from the LLC are simply ignored since a valid copy of these blocks already exists in the main memory. This modification is significantly beneficial for performance.

[3] Blocks with modified data.

Algorithm 11 Flat ORAM Initialization.

```
1:  procedure INITIALIZE( N, B, P )
2:      PosMap := {⊥}^N                          ▷ Empty Position Map.
3:      OccMap := {0}^P                          ▷ Empty Occupancy Map.
4:      for j ∈ {1, ··· , N} do
5:          loop
6:              r ← UNIFORMRAND(1, ··· , P)
7:              if OccMap[r] == 0 then           ▷ If vacant
8:                  OccMap[r] := 1               ▷ Mark Occupied.
9:                  PosMap[j] := r               ▷ Record position.
10:                 break
11:             end if
12:         end loop
13:     end for
14: end procedure
```

6.3.2 Basic Algorithm

Let N be the total number of logical data blocks that we want to securely store in our ORAM, which is implemented on top of a DRAM; and let each data block be of size B bytes. Let P be the number of physical blocks that our DRAM can physically store, i.e. the DRAM capacity (where $P \geq N$).

Initial Setup: Algorithm 11 shows the setup phase of our scheme. Two null-initialized arrays PosMap and OccMap, corresponding to position and occupancy map of size N and P entries respectively, are allocated. For now, we assume that both PosMap and OccMap reside on-chip in the trusted memory to which the adversary has no access. However, since these arrays can be quite large and the trusted memory is quite constrained, we later on show how this problem is solved. Initially, since all physical blocks are empty, each OccMap entry is set to 0. Now, each logical block is mapped to a uniformly random physical block, i.e. PosMap is initialized, while avoiding any collisions using OccMap. The OccMap is updated along the way in order to mark those physical locations which have been assigned to a logical block as 'occupied'. Notice that in order to minimize the probability of collision, P should be sufficiently larger than N, e.g. $P \approx 2N$ gives a 50% collision probability.

Reads: The procedures to read/write a data block corresponding to the virtual address a from/to the ORAM are shown in Algorithm 12. A read operation is straightforward as it does not need to be obfuscated.

Algorithm 12 Basic Flat ORAM considering all PosMap and OccMap is on-chip. Following procedures show reading, writing and eviction of a logical block a from the ORAM.

```
1: procedure ORAMREAD(a)
2:     s := PosMap[a]                                   ▷ Lookup position
3:     data := Dec_K(DRAMREAD(s))
4:     return (s, data)                                 ▷ Position is also returned.
5: end procedure
```

```
1: procedure ORAMWRITE(a, s_old, data)
2:     Stash := Stash ∪ {(a, s_old, data)}              ▷ Add to Stash
3:     return
4: end procedure
```

```
1: procedure EVICTSTASH
2:     (a, s_old, data) ← Stash                         ▷ Read from Stash
3:     loop
4:         s_new ← UNIFORMRAND(1, ··· , P)
5:         if OccMap[s_new] == 0 then                   ▷ If vacant
6:             OccMap[s_new] := 1                       ▷ Mark as Occupied.
7:             OccMap[s_old] := 0                       ▷ Vacate old block.
8:             PosMap[a] := s_new                       ▷ Record position.
9:             DRAMWRITE(s_new, Enc_K(data))
10:            Stash := Stash \ {(a, s_old, data)}
11:            break
12:        else                                         ▷ If occupied.
13:            data' := Dec_K(DRAMREAD(s_new))
14:            DRAMWRITE(s_new, Enc_K(data'))
15:        end if
16:    end loop
17: end procedure
```

The PosMap entry for the logical block a is looked up, and the encrypted data is read through normal DRAM read. The data is decrypted and returned to the LLC along with its current physical position s. The location s is stored in the tag array of the LLC, and proves to be useful upon eviction of the data from the LLC.

Writes: Since write operations should be non-blocking in a secure processor setting, the ORAM writes are performed in two steps. Whenever a data block is evicted from the LLC, it is first simply added to the stash queue, without incurring any latency. While the processor moves on to computing on other data in its registers/caches, the ORAM controller then works in the background to evict the block from stash to the DRAM. A block a to be written is picked from the stash, and a new uniformly random physical position $s_{\texttt{new}}$

is chosen for this block. The OccMap is looked up to determine whether the location $s_{\texttt{new}}$ is vacant. If so, the write operation proceeds by simply recording the new position $s_{\texttt{new}}$ for block a in PosMap, updating the OccMap entries for $s_{\texttt{new}}$ and $s_{\texttt{old}}$ accordingly, and finally writing encrypted data at location $s_{\texttt{new}}$. Otherwise if the location $s_{\texttt{new}}$ is already occupied by some useful data block, the probability of which is N/P, the existing data block is read, decrypted, re-encrypted[4] under probabilistic encryption and written back. A new random position is then chosen for the block a and the above mentioned process is repeated until a vacant location is found to evict the block. Notice that storing $s_{\texttt{old}}$ along with the data upon reads will save extra ORAM accesses to lookup $s_{\texttt{old}}$ from the recursive PosMap (cf. Section 6.3.5).

6.3.3 Avoiding Redundant Memory Accesses

The fact that the adversary cannot see read accesses allows Flat ORAM to avoid almost all the redundancy incurred by a fully functional ORAM (e.g. Path ORAM). Instead of reading/writing a whole path for each read/write access as done in Path ORAM, Flat ORAM simply reads/writes only the desired block directly given its physical location from the PosMap. This is fundamentally where the write-only ORAMs (i.e. HIVE [30], Flat ORAM) get the performance edge over the fully functional ORAMs. However, the question arises whether Flat ORAM is still secure after eliminating the redundant accesses.

6.3.4 Security

Privacy: Consider any two logical write-access sequences O_0 and O_1 of the same length. In EVICTSTASH procedure (cf. Algorithm 12), a physical block chosen uniformly at random out of P blocks is *always* modified regardless of it being vacant or occupied. Therefore, the write accesses generated by Flat ORAM while executing *either* of the two logical access sequences O_0 and O_1 will update memory locations uniformly at random throughout the whole physical memory space. As a result, an adversary monitoring these updates cannot distinguish between *real* vs. *dummy* blocks, and in turn the two sequences O_0 and O_1 seem computationally indistinguishable.

[4] We assume that the encryption/decryption algorithms Enc_K/Dec_K implement probabilistic encryption, e.g. AES counter mode, as done in prior works [20, 61].

Furthermore, notice that in Path ORAM the purpose of accessing a whole path instead of just one block upon each read and write access is to prevent *linkability* between a write and a following read access to the same logical block. In Flat ORAM's model, however, since the adversary cannot see the read accesses at all, therefore the linkability problem would never arise as long as each logical data block is written to a new random location every time it is evicted (which is guaranteed by Flat ORAM algorithm). Although HIVE proposes a constant k-factor redundancy upon each data write, we argue that it is unnecessary for the desired security as explained above, and can be avoided to gain performance.

Hence, the basic algorithm of Flat ORAM presented above guarantees the desired *privacy* property of our write-only ORAM (cf. Definition 6.1.1).

Integrity: Next, we move on to making the basic Flat ORAM practical for a real system. The main challenge is to get rid of the huge on-chip memory requirements imposed by PosMap and OccMap. While addressing the PosMap management problem, we discuss an existing efficient memory integrity verification technique from Path ORAM domain called *PMMAC* [61] (cf. Section 6.5.2) which satisfies our *integrity* definition (cf. Definition 6.1.2).

Stash Management: Another critical missing piece is to prevent the unlikely event of stash overflow for a small constant sized stash, as such an event could break the privacy guarantees offered by Flat ORAM. We completely eliminate the possibility of a stash overflow event by using a proven technique called *Background Eviction* [20]. We present a detailed discussion about the stash size under Background Eviction technique in Section 6.4.4.

6.3.5 Recursive Position Map & PLB

In order to squeeze the on-chip PosMap size, a standard recursive construction of position map [56] is used. In a 2-level recursive position map, for instance, the original PosMap structure is stored in another set of data blocks which we call a *hierarchy* of position map, and the PosMap of the first hierarchy is stored in the trusted memory on-chip (Figure 6.4.1). The above trick can be repeated, i.e., adding more hierarchies of position map to further reduce the final position map size at the expense of increased latency. Notice

that all the position map hierarchies (except for the final position map) are stored in the untrusted DRAM along with the actual data blocks, and can be treated as regular data blocks; this technique is called Unified ORAM [61].

Unified ORAM scheme reduces the performance penalty of recursion by caching position map ORAM blocks in a Position map Lookaside Buffer (PLB) to exploit locality (similar to the TLB exploiting locality in page tables). To hide whether a position map access hits or misses in the cache, Unified ORAM stores both data and position map blocks in the same binary tree. Further compression of PosMap structure is done by using Compressed Position Map technique discussed in section 6.5.1.

6.3.6 Background Eviction

Stash (cf. Section 6.3.1) is a small buffer to temporarily hold the *dirty* data blocks evicted from the LLC/-PLB. If at any time, the rate of blocks being added to the stash becomes higher than the rate of evictions from the stash, the blocks may accumulate in stash causing a stash overflow. Background eviction [20] is a proven and secure technique proposed for Path ORAM to prevent stash overflow. The key idea of background evictions is to temporarily stop serving the real requests (which increase stash occupancy) and issue background evictions or so-called *dummy accesses* (which decrease stash occupancy) when the stash is full.

We use background eviction technique to eliminate the possibility of a stash overflow event. When the stash is full, the ORAM controller suspends the read requests which consequently stops any write-back requests preventing the stash occupancy to increase further. Then it simply chooses random locations and, if vacant, evicts the blocks from the stash until the stash occupancy is reduced to a safe threshold. The probability of a successful eviction in each attempt is determined by the DRAM *utilization* parameter, i.e. the ratio of occupied blocks to the total blocks in DRAM. In our experiments, we choose a utilization of $\approx 1/2$, therefore each eviction attempt has $\approx 50\%$ probability of success. Note that background evictions essentially push the problem of stash overflow to the program's termination channel. Configuring the DRAM utilization to be less than 1 guarantees the termination, and we demonstrate good performance for a utilization of $1/2$ in Section 5.3. Although it is true that background eviction may have different effect on the total runtime for different applications, or even for different inputs for the same application which leak the information about

data locality etc. However, the same argument applies to Path ORAM based systems, and also for other system components as well, such as enabling vs. disabling branch prediction or the L3 cache etc. Protecting any leakage through the program's termination time is out of scope of this work (cf. Section 6.1).

6.3.7 Periodic ORAM

As mentioned in our adversarial model, the core definition of ORAM [18] do not leakage over ORAM timing or termination channel (cf. Section 6.1). Likewise, the fundamental algorithm of Flat ORAM (Algorithm 12) does not target to prevent these leakages. Therefore in order to protect the ORAM timing channel, we adapt the Flat ORAM algorithm to issue periodic ORAM accesses, while maintaining its security guarantees.

In the literature, periodic variants of Path ORAM have been presented [10] which simply always issue ORAM requests at regular periodic intervals. However, under write-only ORAMs, such a straightforward periodic approach would break the security as explained below. Since in write-only ORAMs, the read requests do not leave a trace, therefore for a logical access sequence of `(Write, Read, Write)`, the adversary will only see two writes occurring at times 0 and $2T$ for T being the interval between two ORAM accesses. The access at time T will be omitted which reveals to the adversary that a read request was made at this time.

To fix this problem, we modify the Flat ORAM algorithm as follows. Among the periodic access, for every real read request to physical block s, another randomly chosen physical block s' is also read. Block s is consumed by the processor, whereas block s' is re-encrypted (under probabilistic encryption) and written back to the same location from where it was read. This would always result in update(s) to the memory after each and every time period T.

Security: A few things should be noted: First, it does not matter whether the location s' contains real or dummy data, because the plain-text data content is never modified but just re-encrypted. Second, writing back s' is indistinguishable from a real write request as this location is chosen uniformly at random. Third, this write to s' does not reveal any trace of the actual read of s as the two locations are totally independent.

We present our simulation results for periodic Flat ORAM in the evaluation section.

6.4 Efficient Collision Avoidance

In sections 6.3.3 and section 6.3.4, we discuss how Flat ORAM outperforms Path ORAM by avoiding redundant memory accesses. However, an immediate consequence of this is the problem of *collisions* which now becomes the main performance bottleneck. A collision refers to a scenario when a physical location s, which is randomly chosen to write a logical block a, already contains useful data which cannot be overwritten. The overall efficiency of such a write-only ORAM scheme boils down to its collision avoidance mechanism. In the following subsections, we discuss the occupancy map based collision avoidance mechanism of Flat ORAM in detail and compare it with HIVE's *inverse position map* based collision avoidance scheme.

6.4.1 Inverse Position Map Approach

Since a read access must not leave its trace in the memory in order to avoid the linkability problem, a naive approach of marking the physical location as 'vacant' by writing 'dummy' data to it upon each read is not possible. HIVE [30] proposes an *inverse position map* structure that maps each physical location to a logical address. Before each write operation to a physical location s, a potential collision check is performed which involves two steps. First the logical address a linked to s is looked up via the inverse position map. Then the regular position map is looked up to find out the most recent physical location s' linked to the logical address a. If $s = s'$ then since the two mappings are synchronized, it shows that s contains useful data, hence a collision has occurred. Otherwise, if $s \neq s'$ then this means that the entry for s in the inverse position map is outdated, and block a has now moved to a new location s'. Therefore, the current location can be overwritten, hence no collision.

HIVE stores the encrypted inverse position map structure in the untrusted storage at a fixed location. With each physical block being updated, the corresponding inverse position map entry is also updated. Since write-only ORAMs do not hide the physical block ID of the updated block, therefore revealing the position of the corresponding inverse position map entry does not leak any secret information.

We demonstrate in our evaluations that the large size of inverse position map approach introduces storage as well as performance overheads. For a system with a block size of B bytes and total N logical blocks,

inverse position map requires $\log_2(N)$ additional bits space for each of the P physical blocks. Crucially, this large size of a single inverse position map entry restricts the total number of entries per block to a small constant, which leads to less locality within a block. This results in performance degradation.

6.4.2 Occupancy Map Approach

In the simplified OccMap based approach, each of the P physical blocks requires just one additional bit to store the occupancy information (vacant/occupied). In terms of storage, this gives $\log_2(N)$ times improvement over HIVE.

Insecurely Managing the Occupancy Map

The OccMap array bits are first sliced into chunks equal to the ORAM block size (B bytes). We call these chunks the OccMap Blocks. Notice that each OccMap block contains occupancy information of $8B$ physical locations (i.e. 8 bits per byte; 1-bit per location). Now the challenge is to efficiently store these blocks somewhere off-chip. A naive approach would be to encrypt OccMap blocks under probabilistic encryption, and store them contiguously in a dedicated fixed area in DRAM. However under Flat ORAM algorithm, this approach would lead to a serious security flaw which is explained below.

If the OccMap blocks are stored contiguously at a fixed location in DRAM, an adversary can easily identify the corresponding OccMap block for a given a physical address; and for a given OccMap block, he can identify the contiguous range of corresponding $8B$ physical locations. With that in mind, when a data block a (previously read from $s_{\texttt{old}}$) is evicted from the stash and written to location $s_{\texttt{new}}$ (cf. Algorithm 12), the old OccMap entry is marked as 'vacant' and the new OccMap entry is marked as 'occupied'; i.e. two OccMap blocks $O_{\texttt{old}}$ and $O_{\texttt{new}}$ are updated. Furthermore, the $s_{\texttt{new}}$ location, which falls in the contiguous range covered by one of the two updated OccMap blocks, is also updated with the actual data – thereby revealing the identity of $O_{\texttt{new}}$. This reveals to the adversary that a logical data block was previously read from some location within the small contiguous range covered by $O_{\texttt{old}}$, and it is now written to location $s_{\texttt{new}}$ (i.e. coarse grained linkability). Recording several such instances of read-write access pairs and linking

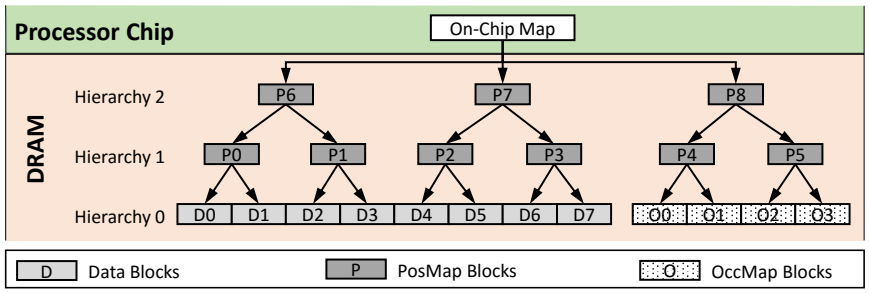


Figure 6.4.1: Logical view of OccMap organization.

them together in a chain reveals the precise pattern of movement of logical block a across the whole memory.

Securely Managing the Occupancy Map

To avoid this problem, we treat the OccMap blocks as regular data blocks, i.e., OccMap blocks are encrypted and also randomly distributed throughout the whole DRAM, and tracked by the regular PosMap. Figure 6.4.1 shows the logical organization of data, PosMap and OccMap blocks. The OccMap blocks are added as 'additional' data blocks at the data hierarchy (Hierarchy 0). Then the recursive position map hierarchies are constructed on top of the complete data set (data blocks and OccMap blocks). Every time an OccMap block is updated, it is mapped to a new random location and hence avoids the linkability problem. We realize that this approach results in overall more position map blocks, however for practical parameters settings (e.g. $128B$ block size, $8GB$ DRAM, $4GB$ working set), it does not result in an additional PosMap hierarchy. Therefore the recursive position map lookup latency is unaffected.

6.4.3 Performance Related Optimizations

Now that we have discussed how to securely store the occupancy map, we move on to discuss some performance related optimizations implemented in Flat ORAM.

Locality in OccMap Blocks

For realistic parameters, e.g. 128 bytes block size, each OccMap block contains occupancy information of 1024 physical locations. This factor is termed as OccMap *scaling factor*. The dense structure of OccMap blocks offers an opportunity to exploit spatial locality within a block. In other words, for a large scaling factor it is more likely that two randomly chosen physical locations will be covered by the same OccMap block, as compared to a small scaling factor. In order to also benefit from such locality, we cache the OccMap blocks as well in PLB along with the PosMap blocks. For a fair comparison with HIVE in our experiments, we also model the caching of HIVE's inverse position map blocks in PLB. Our experiments confirm that the OccMap blocks show a higher PLB hit rate as compared to HIVE's inverse position map blocks cached in PLB in the same manner. The reason is that, for the same parameters, the scaling factor of inverse position map approach is just about 40 which results in a larger size of the data structure and hence more capacity-misses from the PLB.

Dirty vs. Clean Evictions from LLC & PLB

An eviction of a block from the LLC where the data content of the block has not been modified is called a *clean* eviction, whereas an eviction where the data has been modified is called *dirty* eviction. In Path ORAM, since all the *read* operations also need to be obfuscated, therefore following a read operation, when a block gets evicted from the LLC, it must be re-written to a new random location *even if its data is unmodified*, i.e. a clean eviction. This is crucial for Path ORAM's security as it guarantees that successive reads to the same logical block result in random paths being accessed. This notion is termed as *read-erase*, which assumes that the data will be erased from the memory once it is read.

In write-only ORAMs, however, since the read access patterns are not revealed therefore the notion of *read-erase* is not necessary. A data block can be read from the same location as often as needed as long as it's contents are not modified. We implement this relaxed model in Flat ORAM which greatly improves performance. Essentially, upon a *clean* eviction from the LLC, the block can simply be discarded since one useful copy of the data is still stored in the DRAM. Same reasoning applies to the clean evictions from the

PLB. Only the *dirty* evictions are added to the stash to be written back at a random location in the memory.

6.4.4 Implications on PLB & Stash Size

Each dirty eviction requires not only the corresponding data block to be updated but also the two related OccMAP blocks which store the new and old occupancy information. In order to relocate these blocks to new random positions, the 'd' hierarchies of corresponding PosMAP blocks will need to be updated and this in turn implies updating their related OccMAP blocks, and so on. If not prevented, this avalanche effect will repeatedly fill the stash implying background evictions which stop serving real requests and increase the termination time. For a large enough PLB with respect to a benchmark's *locality*, most of the required OccMAP blocks during the benchmark's execution will be in the PLB. This prevents the avalanche effect most of the time (as our evaluation shows) since the OccMAP blocks in PLB can be directly updated.

Even if all necessary OccMAP blocks are in PLB, a dirty eviction still requires the data block with its d PosMAP blocks to be updated. Each of these $d+1$ blocks is successfully evicted from the stash with probability $1/2$, determined by the DRAM utilization, on each attempt. The probability that exactly m attempts are needed to evict all $d+1$ blocks is equal to $\binom{m-1}{d}/2^m$. This probability becomes very small for m equal to a small constant c times $d \log d$. If the *dirty eviction rate* (per DRAM access) is at most $1/c$, then the stash size will mostly be contained to a small size (e.g., 100 blocks for $d = 4$) so that additional background eviction which stops serving real requests is not needed.

Notice that the presented write-only ORAM is not asymptotically efficient: In order to show at most a constant factor increase in termination time with overwhelming probability, a proper argument needs to show a small probability of requiring background eviction which halts normal execution. An argument based on M/D/1 queuing theory or 1-D random walks needs the OccMap to be always within the PLB and this means that the effective stash size as compared to Path ORAM's stash definition includes this PLB which scales linearly with N and is not $O(logN)$.

6.5 Adopting More Existing Tricks

Here we discuss a few more architectural optimizations from the Path ORAM paradigm which can be flawlessly incorporated and are implemented in Flat ORAM for further improvements and features.

6.5.1 Compressed Position Map

The recursive position map for a total of N logical data blocks creates $\lceil \log_b(N) \rceil$ hierarchies of position map. Here b represents the number of positions stored in one PosMap block, and is called *PosMap scale factor*. A higher value of PosMap scale factor would result in less number of PosMap hierarchies and hence yield better performance.

To achieve this goal, Compressed Position Map [61] has been proposed, which results in less PosMap hierarchies than uncompressed PosMap. The basic idea is to store a monotonically increasing counter in the PosMap entry for each logical data block. This counter along with the block's logical address is used as a 'seed' to a keyed pseudo-random function in order to compute a random position for the block. Every time a block is to be written, its PosMap counter is first incremented so that a new random position is generated by the pseudo-random function for the block. To compress these counters to a feasible size, [61] presents an optimization using a big *group counter* and several small *individual counters* per PosMap block. We refer the readers who might be interested in more details to the above citation.

We tweak the compressed PosMap technique for Flat ORAM. The key modification is that the counter for any block to be evicted is incremented even upon unsuccessful eviction attempts, i.e. even if a collision is detected. It is important because otherwise the pseudo-random function will generate the same random location over and over which is already occupied, and hence the block will never be evicted.

6.5.2 Integrity Verification (PMMAC)

Flat ORAM also implements an efficient memory integrity verification technique termed as PosMap MAC (PMMAC) [61]. PMMAC leverages the per-block counters of compressed PosMap to perform MAC[5] checks on the data upon reads. Suppose a logical block a has a counter c, then upon writes, the ORAM controller computes a MAC $h = \mathsf{MAC}_K(a \mid\mid c \mid\mid data)$ using the secret key K and writes the tuple $(h, data)$ to the DRAM. Upon reads, the potentially tampered data tuple $(h^*, data^*)$ is read. The ORAM controller recomputes $h = \mathsf{MAC}_K(a \mid\mid c \mid\mid data^*)$ and checks whether $h = h^*$. If so, the data integrity is verified. Also, since the counter is incremented upon every write, the freshness of the data is also verified, i.e. integrity check guarantees that the most recently written data has been read.

6.6 Experimental Evaluation

6.6.1 Methodology

We use Graphite [63] to model different ORAM schemes in all our experiments. Graphite simulates a tiled multi-core chip. The hardware configurations are listed in Table 6.6.1. We assume there is only one memory controller on the chip, and all ORAM accesses are serialized. The DRAM in Graphite is simply modeled by a flat latency. The 16 GB/s is calculated assuming a 1 GHz chip with 128 pins and pins are the bottleneck of the data transfer.

We use Splash-2 [31], SPEC06 [32] and two OLTP database management system (DBMS) [33] workloads namely YCSB [64] and TPCC [65] to evaluate our Flat ORAM scheme (flat_oram) against various baselines. Three baseline designs are used for comparison: the insecure baseline using normal DRAM (dram), the state of the art Path ORAM with dynamic prefetching [48] (path_oram) and an adaptation of the write-only ORAM scheme from HIVE (hive) in the context of secure processor architectures with several additional optimizations. For all ORAM schemes, we enable the PLB, the compressed position map and integrity verification. The default parameters for ORAM schemes are shown in Table 6.6.1. Unless otherwise

[5] Message Authentication Code (MAC), e.g. a keyed cryptographic hash, is a small piece of information to verify the authenticity of a message/data.

Table 6.6.1: System Configuration.

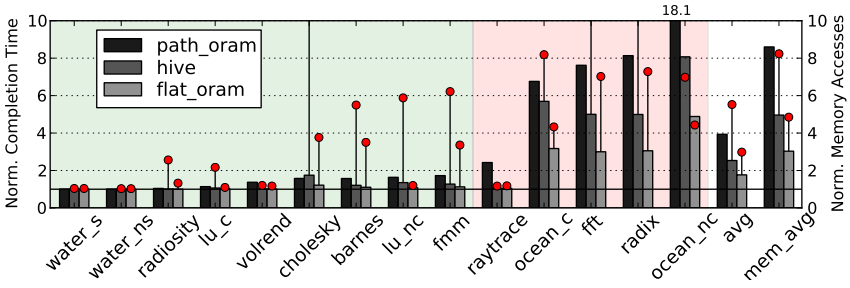
Secure Processor Configuration	
Core model	1 GHz, in order core
Total Cores	4
L1 I/D Cache	32 KB, 4-way
Shared L2 cache	512 KB per tile, 8-way
Cacheline (block) size	128bytes
DRAM bandwidth	16 GB/s
Conventional DRAM latency	100 cycles
Default ORAM Configuration	
ORAM Capacity	8 GB
Working Set Size	4 GB
Number of ORAM hierarchies	4
ORAM Block size	128 Bytes
PLB Size	32kB
Stash Size	100 Blocks
Compressed PosMap & Integrity	Enabled

stated, all the experiments use these ORAM parameters.

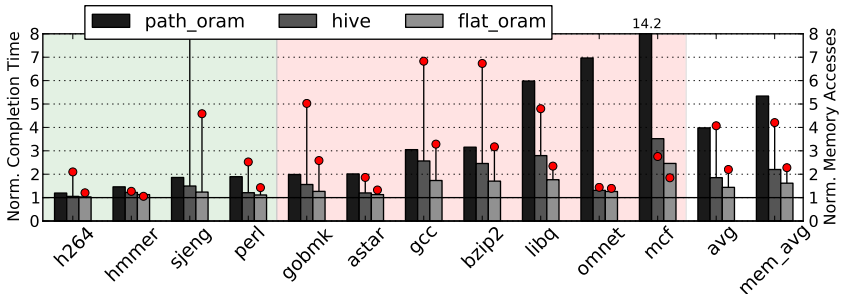
6.6.2 Performance Comparison

Although all ORAM schemes incur performance slowdown over DRAM, however it is important to note that this slowdown is proportional to the memory intensiveness of the application. Memory bound applications suffer from higher performance degradation than compute bound applications. Figure 6.6.1a, Figure 6.6.1b and Figure 6.6.1c show normalized completion times (shown by solid bars) of Splash2, SPEC06 and DBMS benchmarks with respect to DRAM. Splash2 and SPEC06 benchmarks are sorted in ascending order of slowdowns over DRAM from left to right. We consider all the benchmarks with less than 2× overhead as *Computation Intensive* benchmarks (plotted over green background) and all those with more than 2× overhead as *Memory Intensive* benchmarks (plotted over red background).

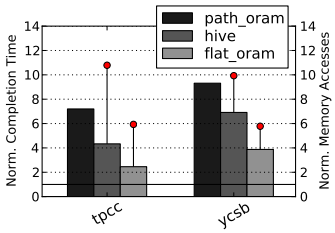
Clearly, Path ORAM incurs the highest overhead, as expected, among all three ORAM schemes because it is a fully functional ORAM which provides higher security. However, the point of presenting this



(a) Splash2



(b) SPEC06

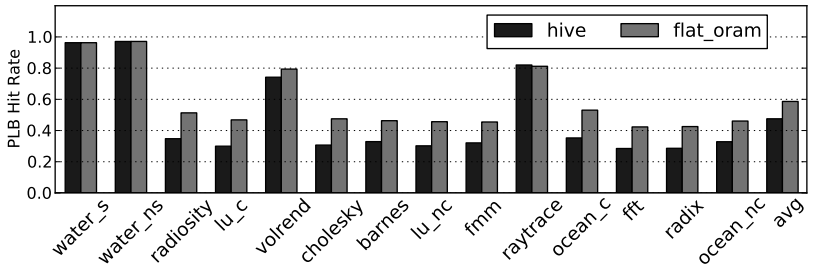


(c) DBMS

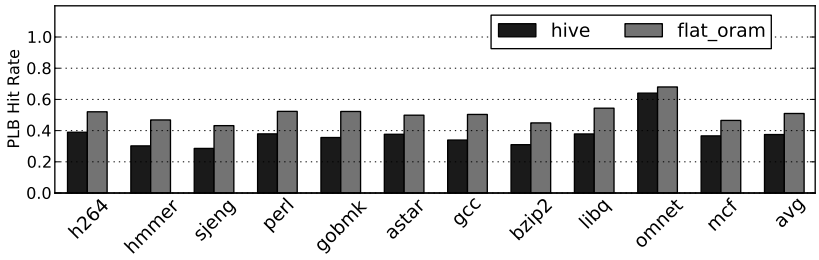
Figure 6.6.1: Normalized Completion Time and Memory Accesses with respect to Insecure DRAM.

comparison is to convince the readers that using Path ORAM for only write-access protection is indeed an overkill when better alternatives (e.g. HIVE, Flat ORAM) exist. On average, Path ORAM incurs about 8.6× slowdown for Splash2 and 5.3× for SPEC06 memory intensive benchmarks (mem_avg). TPCC and YCSB incur 7.2× and 9.3× slowdowns respectively.

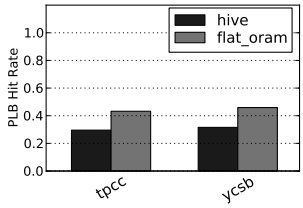
HIVE also shows significant performance degradation compared to Flat ORAM for memory intensive benchmarks (ocean_contiguous, ocean_non_contiguous, mcf, tpcc, ycsb). The average slowdown of HIVE adaptation for memory bound Splash2 and SPEC06 workloads even after several additional optimizations is 5× and 2.2× respectively. Whereas Flat ORAM outperforms HIVE by up to 50% performance gain on



(a) Splash2



(b) SPEC06

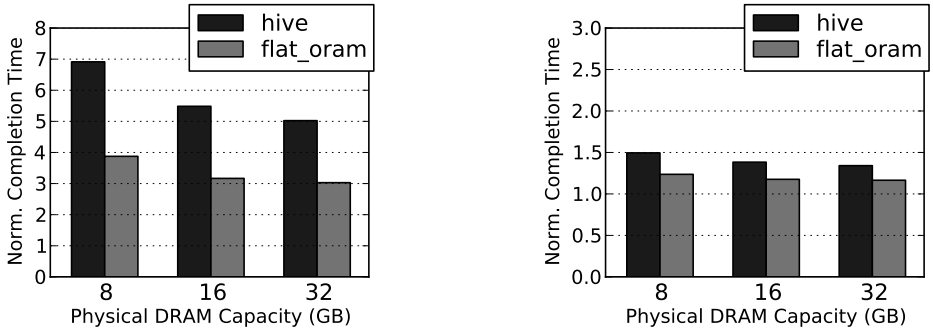


(c) DBMS

Figure 6.6.2: Overall PLB Hit Rate (PosMap blocks and OccMap blocks).

average, having respective average slowdowns of 2× and 1.6×. For DBMS, the performance gain of Flat ORAM over HIVE approach up to 75%.

This performance gap is primarily because the inverse position map approach of HIVE results in significantly increased number of additional DRAM accesses. Figure 6.6.1 also shows normalized total number of DRAM accesses w.r.t. insecure DRAM system (shown by red markers) for HIVE and Flat ORAM. These numbers include both the DRAM accesses issued to serve regular ORAM requests and also the ones caused by background evictions (cf. Section 6.3.6). The normalized access count for Path ORAM is around 200 on average, and is not shown on the plots. It can be seen that HIVE issues 8.2 and 4.2 DRAM accesses



(a) ycsb (b) sjeng

Figure 6.6.3: Sweep Physical DRAM Capacity.

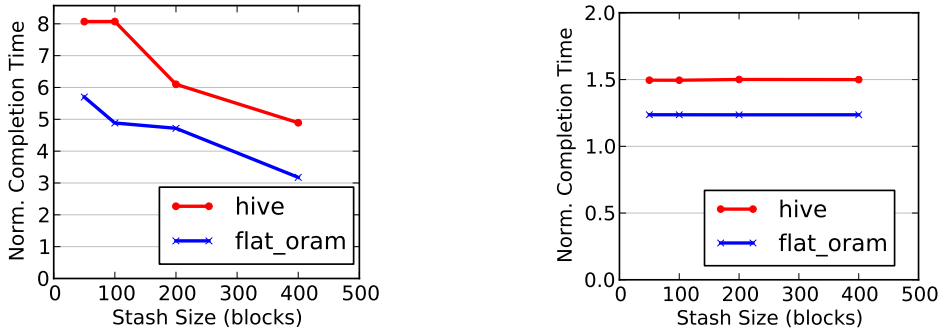
as opposed to Flat ORAM's 4.8 and 2.3 accesses on average for each request issued by the processor for memory intensive Splash2 and SPEC06 workloads respectively.

The reason for higher number of DRAM accesses from HIVE can be found in Figure 6.6.2 which shows the overall PLB hit rate of both HIVE and Flat ORAM. The large memory footprint of the HIVE's inverse position map structure results in overall more data being inserted into the PLB and hence translates into higher number of evictions from PLB. Consequently the ORAM controller experiences higher number of PLB misses and issues relatively higher number of DRAM accesses. Whereas the dense structure of OccMap offers a smaller memory footprint, thus causing less PLB evictions and exhibiting better locality (cf. Section 6.4.3) which directly translates into performance gain.

The normalized number of DRAM accesses is proportional to the energy consumption of the memory subsystem. I.e., a higher number of DRAM accesses would result in more energy consumption. On average, Flat ORAM saves up to 80% energy over HIVE for various workloads.

6.6.3 Sensitivity Study

In this section, we will study how different parameters in the system affect the performance of write-only ORAMs.



(a) ocean_non_contiguous

(b) sjeng

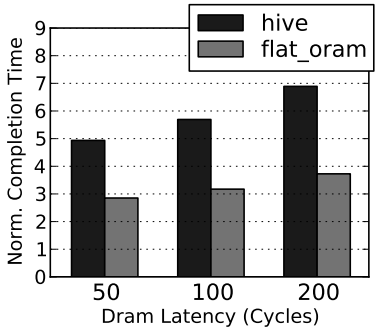
Figure 6.6.4: Sweep Stash Size.

DRAM Utilization: When a block needs to be written to the DRAM, a random position is chosen and if that location is vacant, the block is written at that location (cf. Section 6.3.2). The probability that a randomly chosen location is 'vacant' is determined by the DRAM utilization, i.e. the ratio of occupied blocks to total blocks in DRAM. In order to study the effect of DRAM utilization, we show the results of various physical DRAM sizes (8, 16, 32*GB*) for a constant working set of 4*GB* in Figure 6.6.3. The resulting DRAM utilizations are 50%, 25% and 12.5% respectively.

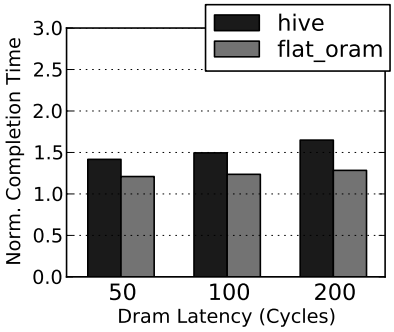
Going from 50% to 25% utilization, memory intensive benchmarks (ycsb) gain performance, as the collisions during write operations are reduced by half. However, the jump from 25% to 12.5% utilization yields little gain because the collision probability of 25% at 16*GB* mark is already too low to be a major performance bottleneck. Notice that HIVE benefits more compared to Flat ORAM from the reduced collisions since it has a much higher collision-penalty. Since less memory intensive benchmarks (sjeng) are not constrained by write operations anyway, lower utilizations do not help much.

Stash Size: As discussed in Section 6.3.6, when the stash occupancy increases than a particular threshold, the ORAM starts performing 'background evictions'. Since background evictions cause the real requests to be suspended temporarily, frequent background evictions cause performance degradation. A larger stash is less likely to become full and thus reduces background eviction rate and improves performance.

In Figure 6.6.4, the stash size is swept for two different benchmarks, one is highly memory intensive

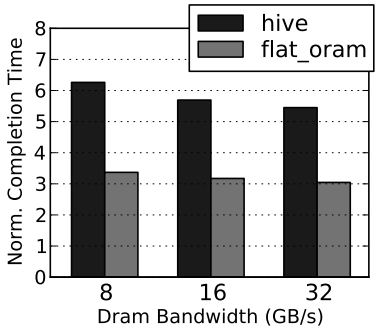


(a) ocean_contiguous

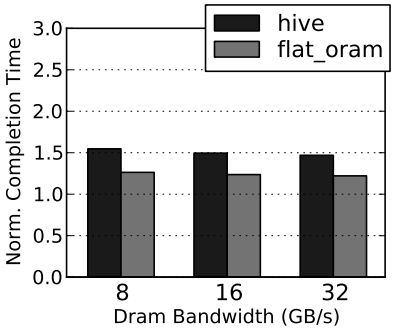


(b) sjeng

Figure 6.6.5: Sweep DRAM latency.



(a) ocean_contiguous



(b) sjeng

Figure 6.6.6: Sweep DRAM bandwidth.

(ocean_non_contiguous) and the other one is significantly less memory bound (sjeng). The memory intensive benchmark benefits from a large stash, as it experiences high background evictions rate at lower stash sizes. The less memory intensive benchmark does not benefit much from increased stash sizes, as it already has a low background evictions rate. In general, Flat ORAM shows significant performance gain over HIVE even at small stash sizes.

DRAM Latency & Bandwidth: In Path ORAM, each ORAM access results in about 200 DRAM accesses on average under typical parameter settings. Most of these accesses can be issued in a burst without waiting for the first data block to arrive, since the addresses are known a priori, e.g. accessing a full path.

Therefore, the DRAM bandwidth becomes the main bottleneck in Path ORAM, whereas the DRAM latency plays less significant role as it is incurred less often.

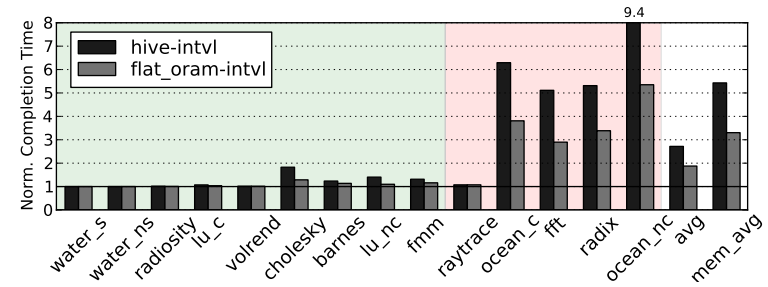
However, the write-only ORAMs under consideration typically only issue less than 10 DRAM accesses per ORAM access (cf. Section 6.6.2). Furthermore, there could be interdependencies within these 10 accesses, e.g., reading an OccMap block to find out if a position is vacant, and then issuing further writes in case a vacant position is found. In such cases, DRAM latency is incurred more often and hence plays more prominent role in the overall performance than the DRAM bandwidth.

This phenomenon is shown in DRAM latency and bandwidth sweep studies in Figure 6.6.5 and Figure 6.6.6 respectively. Memory intensive benchmarks (ocean_contiguous) are more sensitive to DRAM latency and experience more performance degradation at higher latencies. On the other hand, compute bound benchmarks (sjeng) are less sensitive to the DRAM latency. Increasing the DRAM bandwidth seems to help only a little as expected and explained in the discussion above.

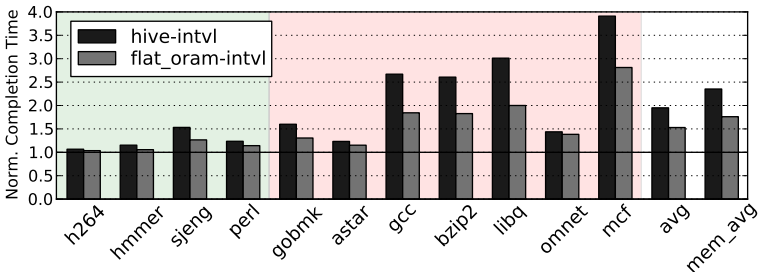
Periodic ORAM: Figure 6.6.7 shows the experimental results of periodic write-only ORAM schemes. The results are normalized to insecure DRAM. The period in terms of number of cycles between two consecutive ORAM accesses is chosen to be 100 cycles. In general, adding periodicity to our ORAM scheme does not significantly hurt performance.

6.7 Chapter Review

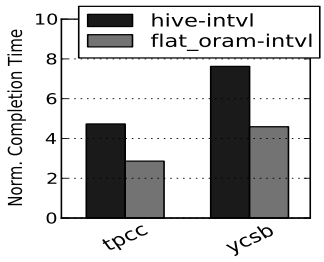
An efficient and practical write-only Oblivious RAM scheme called *Flat ORAM* for secure processor architectures is proposed. It is the first write-only ORAM with a concrete implementation in secure processors domain. The implementation details are discussed and the design space is comprehensively explored. On memory intensive Splash-2 and SPEC06 benchmarks, Flat ORAM only incurs on average 3× and 1.6× slowdown respectively. Compared to a closest related work in the literature, Flat ORAM offers up to 75% higher performance and 80% energy savings.



(a) Splash2



(b) SPEC06



(c) DBMS

Figure 6.6.7: Periodic ORAM accesses. Normalized w.r.t. Insecure DRAM. ORAM Period = 100 cycles.

Chapter 7

Conclusion

This thesis proposes novel architectural primitives for secure processors that address a few of their crucial security vulnerabilities. Specifically, for detection of possible hardware Trojans, a rigorous framework and an associated tool HaTCh is presented that not only detects a small set of publicly known Trojans, but also detects an exponentially large class of deterministic hardware Trojans, while offering provably negligible false negative rate and controllable false positive rate for the corresponding Trojans class. In order to cater for the huge performance penalties incurred by Oblivious RAM (ORAM) used to prevent privacy leakage, an ORAM prefetcher is proposed that detects and exploits data locality in the programs at run time, yielding significant performance gain. Furthermore, to address the weaker adversaries which can only monitor memory write-access patterns, an efficient write-only ORAM scheme has been proposed which avoids the unnecessary overheads of fully functional ORAM schemes, and substantially outperforms the closest existing write-only ORAM in the literature.

This thesis opens up several interesting research directions for future. In hardware Trojan detection domain, would it be possible to use HaTCh as an alternative of *proof carrying hardware* such that the human effort required to develop the proofs can be avoided by leveraging HaTCh and other CAD tools? Similarly, how to design an ORAM prefetcher that not only works for unit stride length but also dynamically adapts any arbitrary stride length for better prefetching, yet its hardware implementation overhead is minimal?

Bibliography

[1] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[2] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *Proceedings of the 1st STC'06*, November 2006.

[3] Trusted Computing Group. TCG Specification Architecture Overview Revision 1.2. http://www.trustedcomputinggroup.com/home, 2004.

[4] David Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[5] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz. Specifying and verifying hardware for tamper-resistant software. In *IEEE S & P*, 2003.

[6] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, 2003.

[7] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9^{th} Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[8] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *ICS*. ACM, June 2003.

[9] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the 32^{nd} ISCA'05*, New-York, June 2005. ACM.

[10] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing; an extended version is located at*

http://csg.csail.mit.edu/pubs/memos/Memo508/memo508.pdf (Master's thesis), pages 3–8, October 2012.

[11] 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.

[12] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[13] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *Design Test of Computers, IEEE*.

[14] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and implementing malicious hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08.

[15] S. Adee. The hunt for the kill switch. *Spectrum, IEEE*.

[16] Yu Liu, Yier Jin, and Yiorgos Makris. Hardware trojans in wireless cryptographic ics: silicon demonstration & detection method evaluation. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press.

[17] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the 11th ASPLOS*, 2004.

[18] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. In *J. ACM*, 1996.

[19] 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 ACM Computer and Communication Security Conference*, 2013.

[20] Ling Ren, Xiangyao Yu, Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the Int'l Symposium on Computer Architecture*, June 2013. Available at Cryptology ePrint Archive, Report 2013/76.

[21] M. Hicks, M. Finnicum, S.T. King, M. Martin, and J.M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *Security and Privacy (SP), 2010 IEEE Symposium on*.

[22] Jie Zhang, Feng Yuan, Lingxiao Wei, Zelong Sun, and Qiang Xu. Veritrust: Verification for hardware trust. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13.

[23] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. FANCI: Identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13.

[24] Mohammad Tehranipoor, Ramesh Karri, Farinaz Koushanfar, and Miodrag Potkonjak. Trusthub. http://trust-hub.org.

[25] Jie Zhang and Qiang Xu. On hardware trojan design and implementation at register-transfer level. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*.

[26] Cynthia Sturton, Matthew Hicks, David Wagner, and Samuel T. King. Defeating uci: Building stealthy and malicious hardware. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 64–77, Washington, DC, USA, 2011. IEEE Computer Society.

[27] Jie Zhang, Feng Yuan, and Qiang Xu. Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.

[28] T. Merin John, S. Kamran Haider, H. Omar, and M. van Dijk. Connecting the Dots: Privacy Leakage via Write-Access Patterns to the Main Memory. *Poster paper at International Symposium on Hardware Oriented Security and Trust (HOST). Available at ArXiv e-prints:* https://arxiv.org/abs/1702.03965, 2017.

[29] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 291–302. Springer, 2002.

[30] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[32] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[33] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment*, 8(3):209–220, 2014.

[34] Syed Kamran Haider, Chenglu Jin, and Marten van Dijk. Advancing the state-of-the-art in hardware trojans design. IEEE International Midwest Symposium on Circuits and Systems, 2017.

[35] Mihir Bellare, Kenneth G Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In *International Cryptology Conference*, pages 1–19. Springer, 2014.

[36] M. Banga and M.S. Hsiao. Trusted rtl: Trojan detection methodology in pre-silicon designs. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*.

[37] Syed Kamran Haider, Chenglu Jin, Masab Ahmad, Devu Shila, Omer Khan, and Marten van Dijk. Advancing the state-of-the-art in hardware trojans detection. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[38] Lang Lin, Wayne Burleson, and Christof Paar. Moles: malicious off-chip leakage enabled by side-channels. In *Proceedings of the 2009 International Conference on Computer-Aided Design*.

[39] 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.

[40] S. Narasimhan, Dongdong Du, R.S. Chakraborty, S. Paul, F.G. Wolff, C.A. Papachristou, K. Roy, and S. Bhunia. Hardware trojan detection by multiple-parameter side-channel analysis. *Computers, IEEE Transactions on*, 62(11):2183–2195, Nov 2013.

[41] D. Forte, Chongxi Bao, and A. Srivastava. Temperature tracking: An innovative run-time approach for hardware trojan detection. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pages 532–539, Nov 2013.

[42] Yier Jin, Nathan Kupp, and Yiorgos Makris. Experiences in hardware trojan design and implementation. In *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*.

[43] ModelSim, Mentor Graphics Inc. www.mentor.com , http://www.model.com.

[44] Jeyavijayan Rajendran, Vivekananda Vedula, and Ramesh Karri. Detecting malicious modifications of data in third-party intellectual property cores. In *Proceedings of the 52nd Annual Design Automation Conference*.

[45] Tony F Wu, Karthik Ganesan, Yunqing Alexander Hu, H-S Philip Wong, Simon Wong, and Subhasish Mitra. Tpad: Hardware trojan prevention and detection for trusted integrated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35, 2016.

[46] Sk Subidh Ali, Rajat Subhra Chakraborty, Debdeep Mukhopadhyay, and Swarup Bhunia. Multi-level attacks: An emerging security concern for cryptographic hardware. In *2011 Design, Automation & Test in Europe*.

[47] Synopsys Inc. http://www.synopsys.com.

[48] Xiangyao Yu, Syed Kamran Haider, Ling Ren, Christopher Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Proram: dynamic prefetcher for oblivious ram. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 616–628. ACM, 2015.

[49] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[50] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.

[51] 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*, CCSW '11, pages 95–100, New York, NY, USA, 2011. ACM.

[52] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 13–24, New York, NY, USA, 2012. ACM.

[53] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[54] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.

[55] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[56] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt*, pages 197–214, 2011.

[57] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[58] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 293–304, New York, NY, USA, 2012. ACM.

[59] Subbarao Palacharla and Richard E Kessler. Evaluating stream buffers as a secondary cache replacement. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society Press, 1994.

[60] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44(5):609–623, 1995.

[61] Christopher Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *Proceedings of the* 20^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[62] Christopher Fletcher, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *Proceedings of the Int'l Symposium On High Performance Computer Architecture*, 2014.

[63] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA*, 2010.

[64] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC'10*, pages 143–154.

[65] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/spec/tpcc_current.pdf, June 2007.

[66] Jacob R. Lorch, James W. Mickens, Bryan Parno, Mariana Raykova, and Joshua Schiffman. Toward practical private access to data centers via parallel oram. *IACR Cryptology ePrint Archive*, 2012:133, 2012. informal publication.

[67] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[68] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 1, pages 56–63. IEEE, 1993.

[69] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44(5):609–623, 1995.

[70] Steven P Vanderwiel and David J Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199, 2000.

[71] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *MICRO*, 2003.

[72] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *MICRO*, 2003.

[73] Blaise Gassend, G Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *HPCA'03*.

[74] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P'15*.

[75] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. In *ACM SIGPLAN Notices*. ACM, 2004.

[76] 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.

[77] Marc Joye and Sung Ming Yen. The montgomery powering ladder. In *CHES'02*.

[78] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*. Springer, 1996.

[79] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48, 2005.

[80] Louis Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In *PKC Workshop*. Springer, 2003.

[81] Sung-Ming Yen, Lee-Chun Ko, SangJae Moon, and JaeCheol Ha. Relative doubling attack against montgomery ladder. In *ICISC*. Springer, 2005.

[82] Damien Aumaitre and Christophe Devine. Subverting windows 7 x64 kernel with dma attacks. *HITB-SecConf Amsterdam*, 2010.

[83] David Maynor. Dma: Skeleton key of computing && selected soap box rants. *CanSecWest: http://cansecwest. com/core05/DMA. ppt*, 2005.

[84] Benjamin Böck and Secure Business Austria. Firewire-based physical security attacks on windows 7, efs and bitlocker. *Secure Business Austria Lab'09*.

[85] Ulf Frisk. Pcileech: Direct memory access attack software.

[86] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. ASIACCS '07.

[87] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *DIMVA*. Springer, 2012.

[88] INC. BPlus Technology. Usb3380 evaluation board.

[89] Robert J McEliece. A public-key cryptosystem based on algebraic coding theory. *Coding Thv*, 4244:114–116, 1978.

[90] Andrey Bogdanov, Marius C Mertens, Christof Paar, Jan Pelzl, and Andy Rupp. Smith-a parallel hardware architecture for fast gaussian elimination over gf (2). In *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS 2006), Conference Records*, 2006.

[91] Erik-Oliver Blass and William Robertson. Tresor-hunt: attacking cpu-bound encryption. In *Proceedings of the 28th ACSAC*. ACM, 2012.

[92] Carsten Maartmann-Moe. Inception. *Break & Enter: http://www. breaknenter. org/projects/inception/[accessed 25 February 2014]*, 2011.

[93] Adam Boileau. Hit by a bus: Physical access attacks with firewire. *Presentation, Ruxcon*, 3, 2006.

[94] Peter Panholzer. Physical security attacks on windows vista. *SEC Consult Vulnerability Lab, Vienna, Tech. Rep*, 2008.

[95] Rafal Wojtczuk and Joanna Rutkowska. Attacking intel trusted execution technology. *Black Hat DC*, 2009, 2009.

[96] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another way to circumvent intel trusted execution technology. *Invisible Things Lab*, 2009.

[97] Syed Kamran Haider and Marten van Dijk. Revisiting definitional foundations of oblivious ram for secure processor implementations. *arXiv preprint arXiv:1706.03852*, 2017.

[98] Lichun Li and Anwitaman Datta. Write-only oblivious ram-based privacy-preserved access of outsourced data. *International Journal of Information Security*, pages 1–20, 2013.

[99] Daniel S Roche, Adam J Aviv, Seung Geol Choi, and Travis Mayberry. Deterministic, stash-free write-only oram. *arXiv preprint arXiv:1706.03827*, 2017.