PHANTOM: Practical Oblivious Computation in a Secure Processor

by

Martin Christoph Maas

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Krste Asanović, Chair
Professor John Kubiatowicz
Professor David Wagner

Spring 2014

PHANTOM: Practical Oblivious Computation in a Secure Processor

Martin Christoph Maas

PHANTOM = **P**arallel **H**ardware to make **A**pplications
Non-leaky **T**hrough **O**blivious **M**emory

In Memory Of
Emil Plamenov Stefanov
(1987-2014)

Abstract

PHANTOM: Practical Oblivious Computation in a Secure Processor

by

Martin Christoph Maas

Master of Science in Computer Science

University of California, Berkeley

Professor Krste Asanović, Chair

Confidentiality of data is a major problem as sensitive computations migrate to the cloud. Employees in a data center have physical access to machines and can carry out attacks that have traditionally only affected client-side crypto-devices such as smartcards. For example, an employee can snoop confidential data as it moves in and out of the processor to learn secret keys or other program information that can be used for targeted attacks.

Secure processors have been proposed as a counter-measure to these attacks – such processors are physically shielded and enforce confidentiality by encrypting all data outside the chip, e.g. in DRAM or non-volatile storage. While first proposals were academic in nature, this model is now starting to appear commercially, such as in the Intel SGX extensions.

Although secure processors encrypt all data as it leaves the CPU, the *memory addresses* that are being accessed in DRAM are still transmitted in plaintext on the address bus. This represents an important source of information leakage that enables serious attacks that can, in the worst case, leak bits of cryptographic keys. To counter such attacks, we introduce Phantom, a new secure processor that obfuscates its memory access trace. To an adversary who can observe the processor's output pins, all memory access traces are computationally indistinguishable (a property known as obliviousness). We achieve obliviousness through a cryptographic construct known as Oblivious RAM (ORAM).

Existing ORAM algorithms introduce a large (100-200×) overhead in the amount of data moved from memory, which makes ORAM inefficient on real-world workloads. To tackle this problem, we develop a highly parallel ORAM memory controller to reduce ORAM memory access latency and demonstrate the design as part of the Phantom secure processor, implemented on a Convey HC-2ex. The HC-2ex is a system that comprises an off-the-shelf x86 CPU paired with 4 high-end FPGAs with a highly parallel memory system.

Our novel ORAM controller aggressively exploits the HC-2ex's high DRAM bank parallelism to reduce ORAM access latency and scales well to a large number of memory channels. Phantom is efficient in both area and performance: accessing 4KB of data from a 1GB ORAM takes 26.2us (13.5us until the data is available), a 32× slowdown over accessing 4KB from regular memory, while SQLite queries on a population database see 1.2-6× slowdown.

Contents

Acknowledgments

PHANTOM was truly a group effort, and as such I want to first and foremost thank my co-authors: Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanović, John Kubiatowicz and Dawn Song. In recognition of the fact that all of the authors contributed to this work, the remainder of the thesis is written in first person plural.

This thesis is based on our paper "PHANTOM: Practical Oblivious Computation in a Secure Processor", which was published at the 20th ACM Conference on Computer and Communications Security (CCS) in November 2013. Using the material in the context of this thesis has been approved by UC Berkeley and all co-authors, and the material has been incorporated into a larger argument (details can be found at the end of the document).

I dedicate this thesis to Emil Stefanov, who passed away shortly after the publication of PHANTOM. It was a pleasure to know Emil and work with him. He was a genuinely kind person, a brilliant researcher and will be dearly missed.

Emil and Elaine did the groundbreaking work on the Path ORAM algorithm that made PHANTOM possible: when they invented the algorithm in 2011, we quickly started talking about the possibility of using it in a secure processor. Together with Mohit, Eric and myself started to work on what would later become PHANTOM as a class project for CS 250 in Fall 2011. As part of this work, Eric and I implemented a simulator to explore high-level trade-offs of Path ORAM implemented in a secure processor setting, but quickly realized that this would not allow us to understand the microarchitectural trade-offs underlying such a secure processor. As part of the class project, we therefore implemented a simple version of the ORAM controller (targeted at an ASIC process rather than an FPGA), and part of this RTL is still at the core of PHANTOM. During the later development of PHANTOM, Mohit also contributed to the code base, in particular with regard to the integration between the ORAM controller and the RISC-V CPU.

During the more than two years from when we started PHANTOM to when we presented the paper at CCS'13, Mohit, Eric and I worked closely on all aspects of PHANTOM, often working late hours in the department writing on code and papers. I want to thank them for all their contributions to this work and the great time we had working on this project.

I also want to thank Emil and Elaine for additional contributions they made to the CCS'13 paper: they performed an empirical analysis of real-world address traces to determine the required size of Path ORAM's stash and ran simulations to show the effectiveness of different reordering schemes (these results are omitted from this thesis since they are not my own contribution). I also want to thank them for countless brainstorming sessions working out design details for PHANTOM.

I would like to thank my advisors Krste Asanović and John Kubiatowicz for guiding me through my first three years of graduate school and advising this project since its inception. They contributed to PHANTOM and the associated papers on every level. Thanks also go to Dawn Song, who advised Emil, Elaine and Mohit and contributed to PHANTOM in an advising capacity as well.

Tremendous thanks is owed to the team that developed the RISC-V processor Phantom is using: among other contributors, this includes Christopher Celio, Henry Cook, Yunsup Lee and Andrew Waterman. Special thanks go to Andrew and Yunsup who spent countless hours helping us with the RISC-V infrastructure.

I would like to thank the instructors from CS250 – John Lazzaro, John Wawrzynek and Brian Zimmer – for their support and feedback during early stages of this project (when we were working on it as a course project for this class).

I would also like to thank Derek Chiou for giving us access to a Convey HC-2ex machine at UT Austin. I further want to thank Yaakoub El Khamra and Laura Timm for supporting us in using this hardware. Special thanks is also owed to Glen Edwards and George Vandegrift from Convey Computer for technical advice and support. In addition, I would like to thank Walid Najjar for giving us access to a Convey HC-2ex at UC Riverside when the UT Austin machine was unavailable, and Victor Hill for supporting us using it.

Thanks is also owed to the anonymous reviewers of our submissions to ISCA '13, CCS '13, CARL '13 and Micro Top Picks '13 for their feedback. Furthermore, I would like to thank the ACM as the publisher of our CCS '13 paper.

Last but not least, I want to thank my girlfriend Yucy and my family for all the support throughout the past years. Without them, none of this would have been possible.

Chapter 1

Introduction

This chapter describes the security challenges for offloading computation to the cloud. It first reviews existing work on secure processors and shows their shortcomings in providing important privacy guarantees. It then shows how these guarantees can be provided by using Oblivious RAM (ORAM), a cryptographic construct that obfuscates memory address traces. Finally, it introduces the Phantom *secure processor which alleviates the aforementioned shortcomings by providing efficient hardware support for ORAM.*

1.1 Security Challenges in Cloud Computing

Enterprises and organizations are increasingly moving computation into the cloud. Cloud computing is predicted to be a $207 billion industry by 2016 [96] and KPMG's global cloud survey among enterprises shows that "70 percent of respondents believe that cloud is delivering efficiencies and cost savings today" [46]. Organizations moving to the cloud span all sectors, from major corporations [6] to NGOs [97] to the US government [23]. While cost savings are a significant driver, other reasons for moving to the cloud include business transformations and increasing business agility [46].

One of the main barriers for adoption of cloud solutions are concerns about the privacy of confidential information. Many organizations operate on highly sensitive data such as financial data, medical data, intellectual property or user data with legally binding privacy requirements. The KPMG Global Cloud Survey cites "data loss and privacy risks" as one of the most pressing concerns when offloading data to the cloud [46] and cloud-based security services are expected to be a $3.1 billion market by 2015 [26].

A particularly difficult security challenge stems from the fact that the cloud provider has physical access to the machines storing and processing a company's privacy-sensitive data. As a result, a malicious employee can readily extract sensitive data from the machines: the importance of such *insider attacks* has been repeatedly demonstrated through examples of rogue employees [47] and thefts of sensitive data at major companies [19]. As recent

high-profile leaks of confidential government information have shown, Top Secret security clearance is not a sufficient guarantee to prevent these leaks [35].

Another concern is security at the cloud provider's data center. As shown in a recent case study [74], data center break-ins are in the realm of possibility and security trade-offs made by the cloud provider may differ from those required by the client. At the same time, the government under whose jurisdiction the cloud provider operates may enforce government-mandated surveillance, potentially without the knowledge of the cloud provider [27]. As these decisions are oftentimes kept secret, the client does not have a way to confirm the data center operator's security-level or even prove that no data was compromised.

Once an attacker has physical access to a machine, they can either run additional code to extract information [95] or extract it directly from the DRAM modules (e.g., using *cold boot attacks* [37]). Many of these attacks are undetectable, in particular if *hardware probes* are deployed, which are known to be actively used by intelligence agencies today [25].

The inability to guarantee privacy of data in the cloud is particularly problematic in cases where the privacy of information is of utmost importance (such as for health records) or where the data gives the company a crucial business advantage (such as financial data used by trading firms to make decisions). In some scenarios, there is even a competitive relationship between the cloud provider and the client. For example, Netflix is hosted on Amazon's EC2 cloud infrastructure, while Amazon is a direct competitor through their Amazon Instant Video service [7].

As a result, cloud customers are becoming increasingly hesitant to offload their computation to third parties (particularly in other countries), due to security concerns. It is predicted that this will lead to losses of $22 billion to $35 billion in revenues for the US cloud industry over the next three years [12]. Overcoming these security challenges is therefore highly important for cloud adoption.

1.2 The Case for PHANTOM

For the past decade, there has been a large body of work in the research community that aims to solve these security challenges. One proposed solution is *Secure Processors* [84, 88, 73, 78]. These are CPUs implemented in tamper-proof hardware which is manufactured by a trusted third party and provides cryptographic means to allow a client to confirm its authenticity before exposing any sensitive information. Outside the processor, data is automatically encrypted, whether in DRAM or non-volatile storage. The advantage of secure processors is that they provide strong privacy guarantees by encrypting data before it leaves the processor, without the prohibitive overheads of multiple orders of magnitude incurred by homomorphic encryption [28], or the maintenance and cost problems associated with making the entire machine (or motherboard) tamper-proof[1].

[1]Individual ASICs are easier to make tamper-proof, since hardware state within the silicon cannot be readily extracted without external interfaces – there has been some recent work on drilling through the silicon [38], but these attacks cannot be deployed during normal data-center operation.

While the first implementations of the secure processor paradigm were academic in nature (such as MIT's AEGIS project [85]), it is increasingly being adopted by industry: IBM's cryptographic co-processors [3] provide a tamper-proof environment for decrypting and processing encrypted data and Intel's upcoming SGX processor extensions [57] protect against memory-based attacks by encrypting an application's address space, making it inaccessible through both physical attack channels and the server's operating system or hypervisor. There are also products emulating this form of hardware support on commodity hardware [66]. All these solutions can be deployed in cloud data centers to ensure privacy of customer data, but other application areas (such as protection of embedded devices or DRM in consumer electronics) are possible as well.

While such platforms prevent an attacker from directly extracting confidential information, they are still prone to side-channel attacks [45]. One of these attacks concerns the accessed memory addresses. Since it is economically infeasible to re-engineer DRAM chips (i.e., the DRAM banks), trusted cloud computing platforms have to rely on off-the-shelf DRAM components. While this does not prevent encryption of all *data* before sending it to memory, it means that *memory addresses* have to be sent in cleartext. As a result, an attacker with physical access to the machine can, e.g., replace the DRAM DIMMs with malicious boards that contain non-volatile memory to log or sample all memory addresses that are being accessed (similar to NVDIMM memory modules that are available today to protect against power outages [4, 58]).

Although memory addresses may appear like a harmless side-channel, they represent an important source of information leakage. Imagine, for example, that a financial company runs a series of checks whenever they are about to sell a stock. In such a case, the addresses accessed by the processor reveal the following information to an attacker:

- The data sets that have been accessed. For example, if the attacker finds out that a particular range of addresses belongs to a particular portfolio, she will know which portfolios are going to be affected by the sale.
- The instruction addresses of the code that is being executed. This can reveal information about the nature of the operation that the client is performing, e.g., whether the company is preparing a sale or a buy.

Depending on the scenario, address information might reveal data such as geolocations, targets of audit or surveillance, vulnerable OS versions or programs (similar to work on OS fingerprinting [36]) and in extreme cases even bits of cryptographic keys [100].

Preventing such information leakage requires making memory address traces computationally indistinguishable, a property known as *obliviousness*. To achieve this goal, this thesis introduces Phantom, a new secure processor that provides not only *data confidentiality* but also *memory trace obliviousness*. In other words, an attacker capable of snooping the memory bus and the DRAM contents cannot learn anything about the secret program memory, not even the memory locations accessed.

1.3 Oblivious RAM

To provide obliviousness, we rely on an algorithmic construct called *Oblivious RAM* (ORAM), initially proposed by Goldreich and Ostrovsky [31], and later improved in numerous subsequent works [63, 34, 16, 49, 33, 77, 83]. Intuitively, ORAM techniques obfuscate memory access patterns through random permutation, reshuffling and reencryption of memory contents. They require varying amounts of *trusted memory* that the adversary cannot observe. To develop a practical ORAM in hardware, we adopt Path ORAM proposed by Stefanov et al. [83] – a simple algorithm with a high degree of memory-access parallelism. Path ORAM builds on a new binary-tree ORAM framework recently proposed by Shi et al. [77].

A concurrent project has also used Path ORAM to propose a secure processor (Ascend [21, 69]); that work focused on optimizing the basic Path ORAM *algorithm* and on a design-space exploration of algorithm parameters using a simple model of a CPU and ORAM controller. In contrast, we focus on the challenges of actually building a practical oblivious system – complete with a CPU, an ORAM controller, and running non-trivial programs like SQLite obliviously on the CPU. The high-level algorithmic optimizations in Ascend are complementary to our algorithmic improvements targeted at Path ORAM's microarchitecture and to our work in designing and implementing a full oblivious system.

1.4 Challenges

Making oblivious processors practical poses several challenges. The first is Path ORAM's significant memory bandwidth overhead: for realistic configurations, it incurs a bandwidth overhead of more than 100× over a non-secure access. We address this problem by relying on an off-the-shelf FPGA platform with a high-bandwidth memory system. However, Path ORAM is difficult to parallelize across multiple memory channels, in particular without introducing security leaks. Furthermore, while our FPGA platform provides the required memory bandwidth, it restricts us to use a slow FPGA for the ORAM controller logic – the ratio of slow logic to high memory bandwidth makes the problem of scaling to a larger number of memory channels even harder. This is representative of the same challenges that have to be solved in a high-performance ORAM system on a custom chip.

To our knowledge, Phantom is the first hardware implementation of ORAM and so many of the microarchitectural challenges have not been previously explored. It would therefore be impossible to model the machine at a more abstract level for simulation without understanding what is involved in an actual implementation. Phantom closes this gap by implementing a full design and exploring the microarchitectural challenges, providing insight into design challenges not visible at an abstract level. In fact, we found a number of microarchitectural details that pose serious challenges in a real implementation but were overlooked in preliminary simulations we performed at earlier stages of the project.

1.5 Contributions

In this thesis, we present Phantom, an oblivious processor that exploits a highly parallel memory system in combination with a novel ORAM controller to implement a practical oblivious system. Specifically, we make the following technical contributions:

1. We present a series of attacks to show how information leakage through memory addresses can be used to extract sensitive informations from applications. This includes discerning different queries to a SQLite database as well as distinguishing different algorithms running on a processor.

2. We introduce an ORAM controller architecture that is very effective at utilizing high DRAM bandwidth – even when implemented on slow FPGA logic. We propose critical improvements of the Path ORAM algorithm, and a deeply pipelined microarchitecture that utilizes 93% of the maximum DRAM bandwidth from 8 parallel memory controllers, while only fetching the minimum amount of data that Path ORAM requires. As a result, Phantom achieves close to the optimal 8× speedup over a baseline design with one memory controller.

3. We build and evaluate Phantom's oblivious memory controller on an FPGA-based computing platform. Using several ORAM configurations, we show that Phantom logic requires only 2% of the LUTs on a Xilinx Virtex 6 FPGA and a single FPGA is sufficient to support an ORAM of 1GB effective size. The Phantom prototype sustains 38,191 full 4KB ORAM accesses per second (150MB/s) to a 1GB ORAM.

4. We integrate the oblivious memory controller with an in-order processor implementing the RISC-V instruction set [92]. We run real-world applications on this processor, including SQLite, and extend our results from the real hardware to different cache sizes by using simulation. Our results show that the oblivious memory controller's overhead translates to 20% to 500% performance overhead for a set of SQLite queries.

1.6 Thesis Organization

Chapter 2 shows a series of attacks that exploit information leakage through the memory address channel to demonstrate that Phantom is solving an important problem. Chapter 3 presents background information on Oblivious RAM (ORAM) research, which is the technique that we use to protect the address channel. This is followed by a high-level introduction of Phantom (Chapter 4), the microarchitectural details of our ORAM implementation (Chapter 5) and how we prototyped it on the Convey HC-2ex heterogeneous computing platform (Chapter 6). Chapter 7 presents a detailed evaluation of Phantom's performance, followed by a summary of related work (Chapter 8) and a discussion of why we believe our work is important (Chapter 9). Chapter 10 then concludes.

Chapter 2

Attacks through Memory Addresses

In this chapter, we demonstrate how memory address traces can be used to attack an application running on a processor, even if the data in memory is encrypted. We first show a simple attack against a SQLite database that demonstrates how address traces reveal information about executed queries. We then show a machine-learning-based attack to discern different algorithms based on instruction address traces.

2.1 Assumptions

For the purpose of these attacks, we assume a very basic setup where we can retrieve full address traces from the stream of memory requests that is issued by the CPU to main memory (i.e., addresses but no temporal information – a CPU could remove temporal information by issuing a memory request every once in a fixed period). This scenario matches an attacker with physical access to the machine who is eavesdropping on the machine's address bus, e.g., through malicious DRAM DIMMs. We assume no additional information, unless explicitly stated otherwise – in particular, we assume no knowledge of DRAM contents, or any state within the CPU or caches (Figure 2.1).

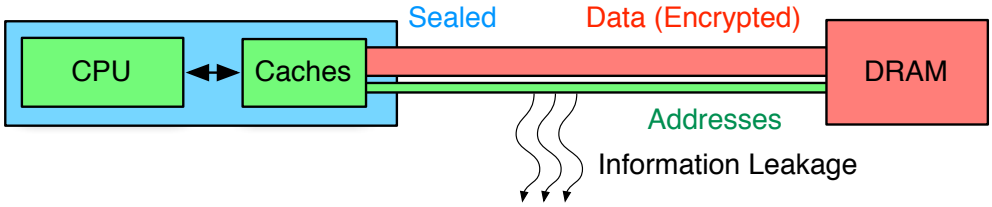


Figure 2.1: The basic attack model - we exploit the indicated information leakage. Red indicates encrypted data while green indicates plain-text. An attacker has physical access to every part of the system except the sealed part indicated in blue.

2.2 Classification of Attacks

Different attack models are conceivable in this scenario. For example, we could assume that we know the code of the application that the victim is running. Based on this knowledge, we could retrace the data flow within the application and reconstruct parts of the confidential data the application is operating on (an instance of this approach has been used to extract bits of cryptographic keys from address traces [100]). It is therefore useful to classify attacks through memory addresses based on whether they assume knowledge of the application's code, the data that it is operating on, or neither.

The case where the data is known but the code is private corresponds to attacks circumventing software protection to learn the functionality of an application based on its memory access pattern. Reconstructing the code of an application is important for reverse engineering programs and learning about attack vectors (e.g., when targeting security-sensitive code in smart cards). It can also be used to extract intellectual property, such as proprietary algorithms that are executed by an application. As a result, software protection was one of the original motivations when Oblivious RAM was first introduced [30].

Recent work has increasingly focused on attacks where an attacker wants to learn about the data being accessed. This is often motivated by the cloud scenario where a cloud provider can learn about the access pattern to data stored in its data centers, e.g., which files are being accessed in a cloud file system [94, 80], which diseases or specialists are being searched for in a medical application [13], or which queries are being executed on a database [2]. Oftentimes, at least part of the code is known in these cases, e.g., an attacker may have access to the code of the file system running on its server, while the data itself is encrypted.

In many cases, neither the data nor the code is known. However, previous work has demonstrated a control flow graph (CFG) fingerprinting technique to identify known pieces of code solely based on the address trace [100], relying on the observation that most applications contain a large portion of reused code – e.g., 39% for the SPEC 2000 benchmarks. This allows an attacker to still correlate memory accesses to program behavior for the reused portions of the code, even if other parts of the program remain unknown.

Some of these different types of attacks are related to other research areas that aim to solve different sets of problems. For example, learning about executing code while the data is known is similar to work on intrusion detection, which aims to detect anomalies in code executing on the machine and can make use of memory traces to achieve this goal. Similarly, learning about a program for which both the code and the data is known has similarities to workload characterization. Table 2.1 summarizes these different scenarios and lists related work for each of them. A comprehensive overview of related work is given in Chapter 8.

For the remainder of this chapter, we will mostly assume an attack model where both code and data are unknown to the attacker – while the CFG fingerprinting approach can often be used to identify reused pieces of code, address space randomization might be used to counter this solution. In the following sections, we first demonstrate how address traces can leak information about both the program that is executing and the data that it is operating on,

Code \ Data	Public	Private
Public	Workload Classification [76]	HIDE [100]
Private	Intrusion Detection [67]	Itai et al. [41], **this work**

Table 2.1: Classification of related work.

even without any advanced processing of the traces. We then show that a machine-learning based approach can extract even more information and automatically recognize different algorithms running on a processor, even when running on different input data.

2.3 Extracting the Data that is Being Accessed

In simple cases, the information leakage through the memory address channel is clearly visible from the trace. For example, in many data structures (such as linked lists, hash tables or binary search trees), the address trace uniquely identifies the accessed element, which in turn leaks information about the executed operations, such as which portfolio is being accessed, or an entry in a medical database. Even without knowing the exact data, this information leakage allows an attacker to link accesses to the same data, and over time learn the layout of the underlying data structure. This is similar to other attacks presented in the past, even though some of these attacks rely on channels other than memory addresses [13, 100].

To demonstrate that the same information leakage occurs in significantly more complex workloads – even in the presence of caches – we simulated a SQLite workload on a processor model with simulated caches. We used the RISC-V ISA simulator [87] for this purpose, since it provides a functional model of the processor we are using for Phantom.

In our experiment, we simulate an in-order processor with a 32KB L1 instruction cache, a 64KB L2 data cache and a 512KB unified L2 cache[1]. We ran SQLite 3.7.15.2 on a publicly available SQLite version of the 2010 US Census Database [20]. This represents a 7.5MB database, which we packaged with the SQLite executable for easier transfer into the simulator. Further, we modified SQLite to redirect all file system accesses into main memory, such that the resulting application never has to access a real file system and runs on bare metal rather than requiring an operating system (we built on `spmemvfs` [53] for this purpose). We also modified SQLite's syscall table to emulate all syscalls within the SQLite executable.

The database we used is small and therefore will hit the cache more often than a bigger data set would. As such, the information leakage we show in this experiment is a conservative estimate – results for more realistically sized data sets are likely to exhibit even more information leakage.

[1]The cache line size for all caches is 128B. The L1 instruction cache is 2-way associative, the L1 data cache 4-way associative and the L2 cache 8-way associative

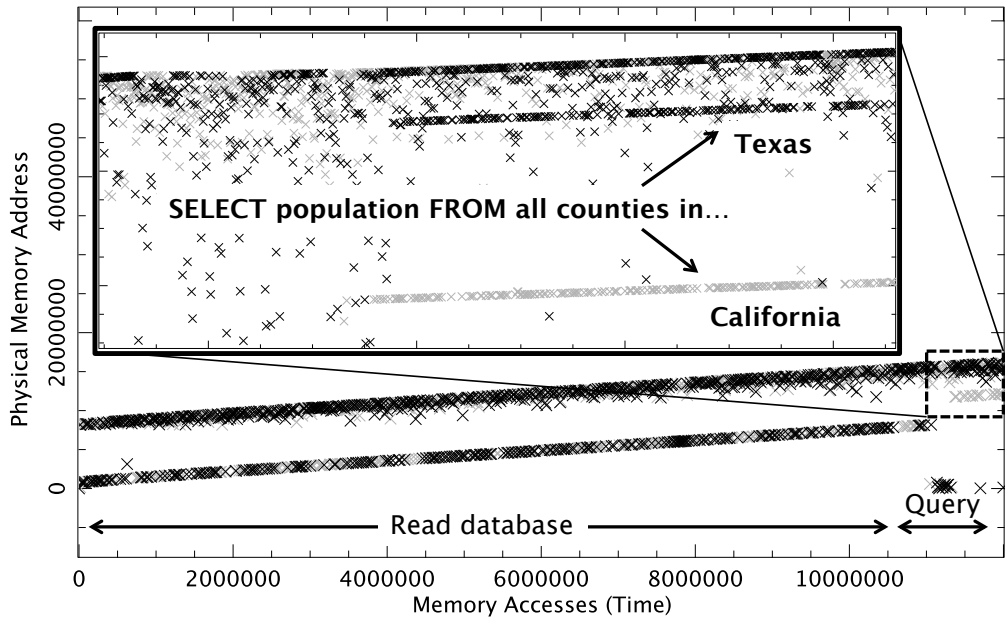


Figure 2.2: Visible information leakage in memory address traces from SQLite, running on a RISC-V processor model with caches. Two queries running on the same SQLite database yield clearly discernible memory accesses.

Figure 2.2 plots the accessed physical memory addresses from last-level cache misses, as produced by our simulator. We ran two different queries:

```
SELECT zctas.zcta,zctas.population_female_total,zctas.population_male_total
    FROM zctas,states_zctas WHERE zctas.id = states_zctas.zcta_id AND
    states_zctas.state_id = <STATE_ID>;
```

where in one case we set `<STATE_ID>` to `'29'` (California) while in the other we set it to `'49'` (Texas). The plot shows that these two queries produce address traces that are identical for the first part (which consists of loading the executable and the database into memory, and copying the database from its `.data` region into a newly allocated area on the heap). However, once the execution of the query commences, the two executions produce visibly different address traces. An attacker could therefore discern the two queries, which represents a significant amount of information leakage about the data that the program is accessing.

2.4 Extracting the Algorithm that is Being Executed

While the previous attack shows how the address trace leaks information about the data that an algorithm is accessing, the following example demonstrates how address traces enable an attacker to learn about the code – and hence the *algorithm* – that is being executed. This

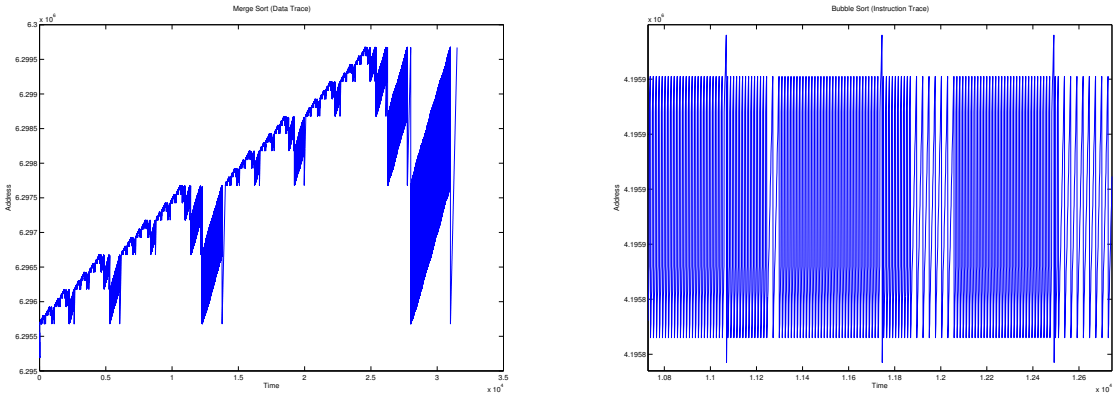


Figure 2.3: An example of a data trace and an instruction trace. The data trace stems from an execution of the merge sort algorithm, the instruction trace from a bubble sort implementation. In both cases, the trace indicates which algorithm is being executed.

is equally important since the execution of a particular algorithm can serve as a fingerprint for a particular function that is running (e.g., to sell stock from a portfolio or to do some routine checks prior to a particular kind of transaction).

In this example, we look at a set of sort algorithms and show what the address traces can tell us about the algorithm that is currently running. Note that for any address trace, we can often discern which parts of it represent instruction fetches and which of them represent accesses to data (based on address ranges in memory). Both of these accesses leak information about the algorithm that is being executed.

To demonstrate this information leakage, consider the two address traces presented in Figure 2.3 (in contrast to the previous experiment, these traces ignore caches and were collected from an x86 executable instrumented using `valgrind` – the precise methodology is described in Section 2.5). The first plot represents a *data trace*, which contains all addresses accessed on the heap of a program running `glibc`'s `sort` function. From this trace alone, it is possible to see that the algorithm performed by this function is merge sort: the trace clearly shows the different phases of the algorithm – sorting two subregions recursively and then merging them in a linear scan through both regions.

Similarly, the second graph presents the *instruction trace* of a bubble sort implementation. The "spikes" show the beginning of the next iteration of the outer loop while the linear accesses in between represent the linear scan through the array. Note that some iterations take longer than others – this represents whether two consecutive elements are being swapped or not and leaks information about the input data. The trace even shows how these swaps often occur in consecutive locations due to entries "bubbling" through the array.

While far from a rigorous analysis, this example intuitively shows that there is a significant amount of statistical information contained in address traces. In reality, this information is not as readily available as in this example, since instruction and data caches hide a signif-

icant portion of the memory accesses and introduce noise. The next section presents work towards statistical methods to extract the information contained in address traces and use it to learn sensitive information about the program that generated them.

2.5 Using Machine Learning to Extract Secrets

While information can sometimes be visually observed from the address traces, most attacks through address traces require a more sophisticated approach. In particular, in a scenario with more complex application mixes and architectures, the address trace may appear random at visual inspection, even if it includes a large amount of information. Simple defenses against information leakage through the address channel (such as HIDE [100] or address space randomization) introduce additional noise as well. We therefore explore statistical techniques to still extract information from the memory address trace, even in cases where this information is not visually apparent. Such techniques can be used to automate attacks and search for relevant information in a trace collected during a physical attack.

Methodology

To make the problem more tractable, we restrict ourselves to identifying the algorithm a victim is executing. This information may allow us to predict the victim's actions (e.g. if it is a financial firm, whether it is about to sell or buy) and learn properties of data (e.g. if the victim is operating on medical records and runs an algorithm specific to a particular disease, we know that it is part of the record). We focus on instruction traces and assume that all addresses are observed (it has been shown that this could be approximated by maliciously causing cache flushes [100]). Modeling the impact of caches is future work.

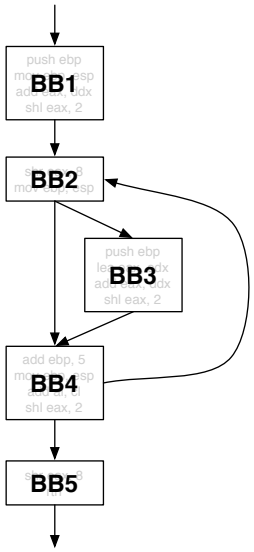
Formally, our attack can be described as follows: we want to capture each operation $O \in \mathbb{O}$ that the victim may perform by a model θ_O such that when we observe an address trace Y, we can determine that it stemmed from O, i.e.

$$\forall O' \in \mathbb{O}.(O' \neq O) \Rightarrow \mathcal{L}(Y|\theta_O) \gg \mathcal{L}(Y|\theta_{O'}).$$

We base our research on instruction traces gathered from implementations of three common algorithms: (i) Merge Sort, (ii) Bubble Sort and (iii) Sparse Matrix Multiplication [42]. These algorithms are similar in complexity but sufficiently different to give meaningful results. To collect our memory traces, we used the open-source `valgrind` [59] tool, which allows instrumentation of binaries. We developed our own Valgrind plug-in (based on the `lackey` example tool included with Valgrind) that allows us to record full instruction or data traces, and to switch tracing on and off through macros in the profiled application. For each of the three algorithms, we collected one trace of training data and one trace of test data stemming from different inputs. The traces were 22,391-44,429 addresses in length.

The Basic Probabilistic Model

Programs can be divided into basic blocks. A basic block is a unit of consecutive instructions with the property that it has a single entry and exit point, i.e. execution will never jump into or away from interior instructions of a basic block. This allows us to model (deterministic) execution of an application on data D as follows:



$$X_t \in \mathbb{N} - \text{Instruction address at time } t$$

$$\pi \in \mathbb{N} - \text{Program entry point}$$

$$p(X_0 = i) = \begin{cases} 1 & \text{if } \pi = i \\ 0 & \text{otherwise} \end{cases}$$

$$p(X_{t+1}|X_t, D) = \begin{cases} 1 & \text{if } X_t \text{ is not the end of basic block and } X_{t+1} = X_t + 1 \\ 0 & \text{if } X_t \text{ is not the end of basic block and } X_{t+1} \neq X_t + 1 \\ q(X_{t+1}|X_t, D) & \text{otherwise, for some unknown but fixed distribution } q \end{cases}$$

The distribution q captures all the program behavior (i.e transfer probabilities between different basic blocks) and can be arbitrarily complex. In order to model program execution, we therefore need a way to approximate q. We do this by dropping D from the equation (since the data is unknown to us) and enforce the Markov property, i.e. $q(X_{t+1}|X_t, D)$ becomes $q(X_{t+1}|X_t)$. For simplicity, we also assume time homogeneity (however, according to results from [76], it may be beneficial to drop this assumption at some later point). In practice, this means that the model does not capture certain changes between program phases.

The second assumption we have to drop is the fixed entry point π. Since we cannot assume that our recorded trace starts precisely at the entry point, we use a distribution over states that we fit to the training data based on maximum likelihood. The resulting model is a time-homogeneous Markov Chain where states represent basic blocks (Figure 2.4). Here, the y_t's represent the observed addresses – the likelihood of observing a particular y_j while in state X_i is captured by the *emission probability* of the model.

The Empirical Model

To confirm the validity of this model, we used the training data to build an empirical model that constructs the Markov Model directly from the training data. In this model, each state of the Markov Chain corresponds to a basic block and the transition probabilities between states are the relative transition frequencies between basic blocks as they occurred in the

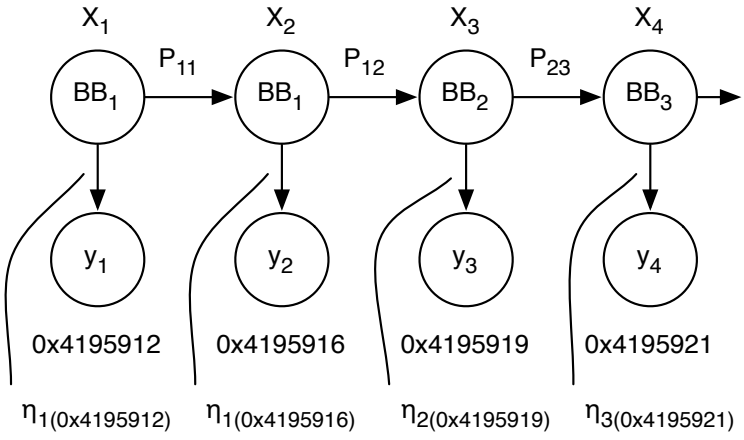


Figure 2.4: The graphical model of the Markov Chain with multinomial emission probabilities that underlies the empirical model.

training data. The emission probabilities of the Markov Chain are modeled as multinomials, with uniform emission probabilities for addresses within the current basic block and zero probability for all addresses outside the block (here, y_1^T is the training address trace):

$$s_i - \text{First address of } BB_i \quad e_i - \text{Last address of } BB_i \quad l_i = e_i - s_i + 1$$

$$Q = \#\ BBs \quad b_{ij} = \sum_{t=1}^{T-1} \mathbb{I}(s_i \leq y_t\ \&\ y_t \leq e_i\ \&\ s_j \leq y_{t+1}\ \&\ y_{t+1} \leq e_j)$$

$$\pi = \vec{1} \cdot (1/Q) \quad P_{ij} = \frac{b_{ij}}{\sum_{k=1}^{Q} b_{ik}} \quad \eta_{ij} = \frac{1}{l_i}\mathbb{I}(s_i \leq j\ \&\ j \leq e_i)$$

Note that the basic block boundaries s_i, e_i are not available to an attacker who only observes the address trace. To construct the empirical model, we extracted them using `BBV`, a Valgrind tool that detects branch instructions to determine basic blocks.

The empirical model correctly matched the test address traces with the algorithm that generated them: after fitting π for each test trace (using the EM algorithm), keeping the other model parameters fixed, the resulting log-likelihood was $-50,012$ for the bubble sort model on the bubble sort trace, $-165,051$ for the merge sort model on the merge sort trace and $-80,657$ for the matrix multiplication model on the matrix multiplication trace. All other likelihoods were zero. This is not surprising, since the multinomial emission probabilities mean that a trace that contains even a single address which didn't occur in the model's training data (a very likely scenario) results in likelihood 0. While this makes the model too fragile for practical use, it shows that the basic approach of modeling execution as a Markov Model works well. We will now show how to refine the model to be more practical.

Model \ Test Data	BSort	MSort	SparseMM
BSort	**-50,590**	-439,390	$-\infty$
MSort	-309,450	**-121,780**	$-\infty$
SparseMM	-326,320	-349,680	**-48,240**

Table 2.2: Results for HMM with Gaussian emission probabilities (Q=50 states). We highlight the maximum likelihood in each column.

The Naïve Hidden Markov Model

The empirical model relied on knowledge of the basic blocks, which the attacker does not have. We therefore need a model that learns the basic blocks instead. A Hidden Markov Model (HMM) achieves this: by fitting the HMM to the training data, it will learn a set of states (basic blocks) and instructions that belong to them (emission probabilities). We fit a Hidden Markov Model with 50 states and multinomial emission probabilities (with one outcome for each address in the output) using the EM algorithm. After fitting the model, we performed the same test as with the empirical model, fitting the test data's prior distribution π and comparing log-likelihoods. The results were similar: all three algorithms were correctly classified (log likelihoods $-4,123$ for bubble sort, $-80,146$ for merge sort and $-21,286$ for sparse matrix multiplication). The other likelihoods were, as before, zero.

Gaussian Emission Probabilities

While the previous approach achieves its goal for our examples, it is impractical: it is too fragile and has too many free parameters (one emission probability for each address, which number in the millions even in small programs). Both problems are solved by replacing the multinomial emission probabilities by Gaussian emission probabilities. Each state i receives a parameter pair (μ_i, σ_i^2), which can be interpreted as the center and the size of each basic block[2]. As expected, the results are not as clear cut as before anymore – resulting in non-zero likelihoods for most pairs – but still classify each algorithm correctly (Table 2.2).

Adapting the EM Algorithm to support code relocation

The above approach fails when the basic blocks are at different locations than in the training data. However, this is a common case as many systems use address space randomization for security reasons. Furthermore, recompilation and small changes to software or compiler lead to changes in addresses as well.

[2]We enforced $\sigma^2 \geq 1$ since otherwise the optimization can result in Gaussians with a very small variance in cases where there are more states than instructions, giving an invalid interpretation (blocks with < 1 instructions) and a positive log likelihood.

Model \ Test Data	BSort	MSort	SparseMM
BSort	-222,090	-325,450	$-\infty$
MSort	-265,350	$-\infty$	$-\infty$
SparseMM	-318,270	-332,490	-172,790

Table 2.3: Results for HMM approach with mean fitting for relocated code.

A simple way to support such *code relocation* would appear to be fitting the means again when fitting the model to the test data, while leaving transition matrix and variances fixed as before. However, running this approach on data that has been shifted by +500 shows that the results do not look very encouraging (Table 2.3). While the method correctly classifies BSort and SparseMM, the distinction is not very clear and the results for MSort appear to be meaningless. We hypothesized that the reason for this is that refitting the mean loses a fundamental property of the model: the relative distances of the means for the basic blocks. Since code relocation should not change these values significantly, a natural solution seems to be not to fit all the μ_i, but instead fit a single global offset M and use a model with means $\mu_i + M$ where the μ_i are fixed.

To find this M, the EM algorithm needs to be adapted. For the E step, every occurrence of μ_i has to be replaced by $\mu_i + M$. For the M step, we need to derive the update for M. In the new model, we need to maximize $\log p(q, y)$, i.e.

$$\sum_{i=1}^{Q} q_0^i \log \pi_i + \sum_{t=0}^{T-1} \sum_{i,j=1}^{Q} q_t^i q_{t+1}^j \log P_{ij} + \sum_{t=0}^{T} \sum_{i=1}^{Q} q_t^i \left\{ -\frac{1}{2} \left(\frac{(y_t - \mu_i) - M}{\sigma_i} \right)^2 \right\}$$

$$\text{Zero-gradient for last term: } M \left(\sum_{t=0}^{T} \sum_{i=1}^{Q} \frac{q_t^i}{\sigma_i^2} \right) = \sum_{t=0}^{T} \sum_{i=1}^{Q} \frac{q_t^i}{\sigma_i^2} (y_t - \mu_i)$$

$$\text{Parameter update: } M \leftarrow \frac{\sum_{t=0}^{T} \sum_{i=1}^{Q} \frac{\gamma_t^i}{\sigma_i^2} (y_t - \mu_i)}{\sum_{t=0}^{T} \sum_{i=1}^{Q} \frac{\gamma_t^i}{\sigma_i^2}}$$

However, it turns out that when applying this algorithm, parameter M barely changes. The reason is that due to the nature of the data, we have a very large number of local maxima (which correspond to addresses in the model being shifted over addresses in the data, whether they are the right addresses or not). This becomes evident when looking at the log likelihood as a function of M (Figure 2.5). A solution to this problem is to get an approximation of the global maximum (e.g. by setting all γ's in the EM algorithm to the same value) and then performing a linear search in both directions. Once the right value of M has been found, the results are the same as for the original algorithm.

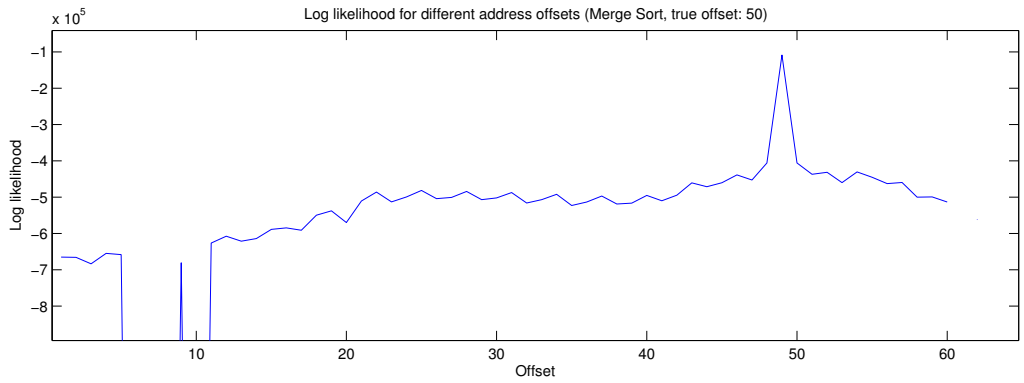


Figure 2.5: Log-likelihood as a function of the global offset M.

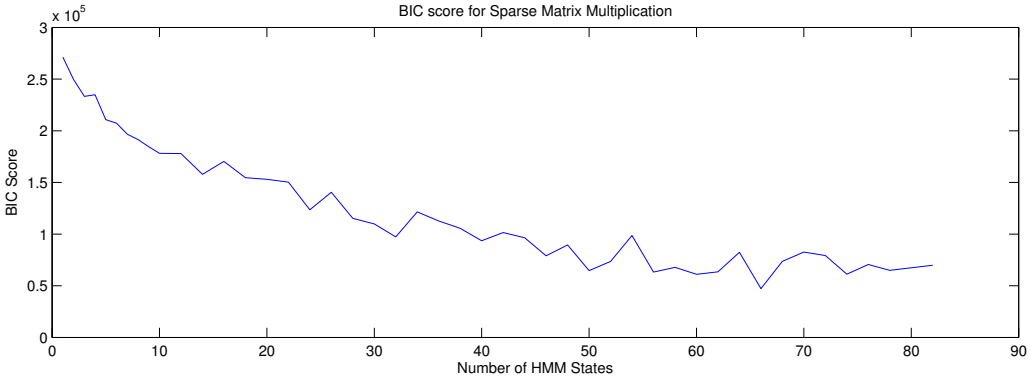


Figure 2.6: BIC score for Sparse Matrix Multiply as a function of the number of states.

Using BIC to Determine the Right Number of States

Choosing the right number of states for the Markov Chain is crucial for the quality of the model in order to avoid overfitting. As explained by Raftery [68], minimizing the BIC score is a good way to determine the right number of states. The BIC score is calculated as $-2L + k \cdot \log n$ where n is the length of the training data, L is the resulting log likelihood and k is the number of free parameters. Notably, the number of free parameters in this case is $k = Q \cdot (Q-1) + 3Q$: $Q(Q-1)$ from the transition matrix, Q from the prior, Q from the means and Q from the variances. Figure 2.6 presents the BIC score for the case of Sparse Matrix Multiplication. According to the BIC score, the best number of states is 66.

An interesting observation is that this number is vastly higher than the number of basic blocks (32 in this case). This could imply overfitting, but might also only mean that it makes the model easier to fit. In principle, additional states do not hurt since a basic block can always be split into two blocks with transition probability one between them.

A Comment on Running Time

Fitting the models to training data (max. 20 iterations of EM) took 2 minutes on a 2011 Macbook Pro, fitting the prior as part of classification (max. 10 iterations) about 20 seconds. All computation was performed in MATLAB.

2.6 Limitations and Future Work

Naturally, this example only scratches the surface of the topic and the presented results can only be considered preliminary. In particular, the following represent some important limitations of the approach, and opportunities for future work.

- **Impact of Caches**: The biggest omission is the impact of caches. In a real-world scenario, this will be a key limiting factor on the quality of classification that can be achieved. Caches can be simulated using Valgrind's `cachegrind` tool and will have the effect that (i) not all addresses will be observed (blurring the transition probabilities) and (ii) addresses will be quantized to cache lines. It remains to be seen whether a pure HMM approach would be sufficient in this case or whether additional measures are required (as in [99]).
- **Large-scale Programs**: The current approach will not scale to programs with large numbers of basic blocks. To achieve this, it would be possible to use an approach similar to [76] and accumulate information about visited basic blocks into *basic block vectors* and use random projections of them to get low-dimensional data to classify.
- **Alternative Models**: HMMs are only one possible approach. A different strategy could use a Naïve Bayes classifier or support vector machines to classify traces based on features such as access frequencies and strides (which could be determined by applying an FFT to the data) or analyzing data traces in addition to instruction traces.

2.7 Summary

In this chapter, we demonstrated that memory address traces contain exploitable information and showed first steps towards using them to learn secrets about workloads running on a machine. Oftentimes, manual inspection of the trace directly yields exploitable information leakage. For more complex scenarios, we saw that HMMs provide a promising model to classify algorithms but also noticed that code relocation and input size pose challenges. The effect of caches should raise further interesting research questions. In summary, this work demonstrates that information leakage on the address bus is a serious concern and work to prevent it is warranted.

Chapter 3

Background on Oblivious RAM

This chapter presents previous research on Oblivious RAM, followed by a detailed review of the Path ORAM Algorithm which is used in this work. Furthermore, it reviews joint work with Stefanov et al. to determine the required size of an ORAM processor's on-chip storage.

3.1 The Need for Oblivious RAM

As shown in Chapter 2, memory address traces from programs running on a processor result in significant information leakage that can be exploited in a number of different ways. As a result, clients with strong requirements for confidentiality will have to protect themselves against this side channel to ensure the confidentiality of information.

For this protection to be trusted, a formal model is required to define what it means for an execution to be free of information leakage through address traces. This model needs to ensure that an attacker cannot learn anything about the logical memory addresses being accessed by the program, even if she can observe the full trace. While some work has attempted to use an intuitive definition of hiding information leakage through address traces [100], such a model is not sufficient to guarantee obliviousness – without formal, provable guarantees, a client can never be sure that her confidentiality requirements are satisfied and that no information leakage occurs, even in corner cases.

Memory Trace Obliviousness (or *obliviousness*) is a formal property that captures the above intuitive requirement. Over the years, the research community has proposed a number of mechanisms (collectively known as *Oblivious RAM* or *ORAM*) to provide the obliviousness property in a number of computing scenarios. In the next sections, we will give a formal definition of obliviousness and review existing research in this area. We will then focus on a recently proposed ORAM algorithm – *Path Oblivious RAM* – that we use in Phantom.

3.2 Definition of Obliviousness

While the concept of memory trace obliviousness has been applied to a multitude of different scenarios, obliviousness was first introduced in the context of a program running on a trusted, sealed CPU and an attacker who wants to learn information about the program or the data it is operating on – while only being able to observe the sequence and timing of accessed addresses in physical memory, not its content. More formally, the obliviousness property for such processors can be defined as follows:

Definition 1. *Let* $A_n(x) = ((a_1, o_1), \ldots, (a_n, o_n))$ $(a_i \in \{0, \ldots, a_{max} - 1\}, o_i \in \{read, write\}$ *for all* i*, where* a_{max} *is the size of the processor's memory in words) be the sequence of memory accesses that a processor* P *produces when run on input* x *(i.e., the program to execute as well as any input data – specifically, this is the entire initial content of the processor's memory) for a duration of* n *memory accesses. We call* P *oblivious if for any inputs* x *and* x'*,* $A_n(x)$ *and* $A_n(x')$ *are computationally indistinguishable for all* $n \in \mathbb{N}$[1].

The most intuitive way to achieve obliviousness is to simply access all of physical memory for every memory access. Hence, for every access, the processor would read all of memory but only use the word that was actually requested. For writes, the processor would have to read and write every memory word, and re-encrypt them using probabilistic encryption (e.g. by adding a randomly generated nonce to each cipher block), to avoid leaking which memory word was written. While this approach is obviously infeasible for performance reasons, it gives a good intuition about how obliviousness works. The goal of research on Oblivious RAM has been to reduce this overhead to acceptable levels, while maintaining the same security guarantee.

3.3 History of Oblivious RAM

Oblivious RAM was first proposed by Goldreich [30] and Ostrovsky [60] who built on work by Pippenger and Fischer [64] on Oblivious Turing Machines (who, interestingly, discussed obliviousness in a completely different context than security). While Goldreich and Ostrovsky were investigating Oblivious RAM for Software Protection [31] – to prevent an attacker from extracting the algorithm executed by a sealed processor – their attack model is very similar to ours, where an attacker with physical access to a CPU can wiretap the address bus, and memory contents are encrypted. Goldreich and Ostrovsky achieved obliviousness with an overhead of $O(\log^3 N)$ per memory access (where N is the size of the oblivious memory in memory words), albeit with very large constant factors [63]. These constant factors made this approach infeasible for practical deployment in most scenarios.

[1] Note that this definition does not consider timing channels (i.e. *when* the n memory accesses occur). This is a source of information leakage orthogonal to obliviousness and will be discussed in Section 3.4. It also ignores active attacks where an attacker can return incorrect responses to memory requests – these can be countered by adding integrity and freshness (Section 3.6).

Numerous projects have since then revisited Oblivious RAM to reduce this performance overhead [34, 61, 49, 93, 32, 82, 77, 29, 14] and show that it is possible to achieve obliviousness with low constant factors for a $O(\text{polylog}(N))$ overhead, i.e., every memory access translates into $O(\text{polylog}(N))$ seemingly random accesses. There have also been a number of projects that apply ORAM techniques to specific scenarios such as oblivious file systems [94, 54, 80], databases [2] and ad serving [9].

While Phantom builds on this line of work, it brings Oblivious RAM back to the roots by applying these techniques to the secure processor scenario for which ORAM was originally designed. Instead of devising a new ORAM scheme, Phantom considers a state-of-the art algorithm and shows how to adapt it for an efficient hardware implementation. As such, it is the first project to present a real microarchitectural implementation of ORAM (Ascend is a concurrent project that also proposes an oblivious processor [21, 69] but is simulation-based and focusses on algorithmic aspects of ORAM rather than microarchitectural details).

3.4 Path Oblivious RAM

Phantom builds on an ORAM scheme called *Path Oblivious RAM* (or *Path ORAM*) which was first publicly introduced by Stefanov and Shi in 2012 [81]. The authors have since characterized the algorithm in more detail and proved security guarantees in subsequent refinement of the work. The final version of Path ORAM was published alongside Phantom at the 20th ACM Conference on Computer and Communications Security (CCS'13)[2] [83].

Intuitively, the Path ORAM algorithm prevents information leakage through memory addresses by reshuffling contents of untrusted memory after each access, such that accesses to the same location cannot be linked (while also probabilistically reencrypting the accessed content at every access). Furthermore, we assume the secure processor has a small amount of trusted memory, which Path ORAM can access without revealing any information to the attacker. This memory is used to keep track of where data resides in untrusted memory. Using the trusted memory, Path ORAM ensures that all that is visible to an attacker is a series of random-looking accesses to untrusted memory.

Path ORAM allows data to be read and written in units called *blocks*. All data stored by an ORAM instance is arranged in untrusted memory as a binary tree structure, each node of which contains space to store a few blocks. When a request is made to the ORAM for a particular block, Path ORAM looks up the block's current location in a table in trusted memory called the *position map*. In the position map, every block is assigned to a particular *leaf node* of the ORAM tree, and the Path ORAM algorithm guarantees an invariant that each block will be resident in one of the nodes along the path from the tree's root to the

[2] Phantom has been under development in parallel to this work, starting from Fall 2011. We have closely collaborated with Stefanov and Shi, who have made significant contributions to this project while working on the algorithm. Among others, they have provided an empirical analysis of the storage requirements for the Path ORAM algorithm (Section 3.5) and conducted experiments to empirically show the significance of the reordering scheme used with Path ORAM (Section 4.6).

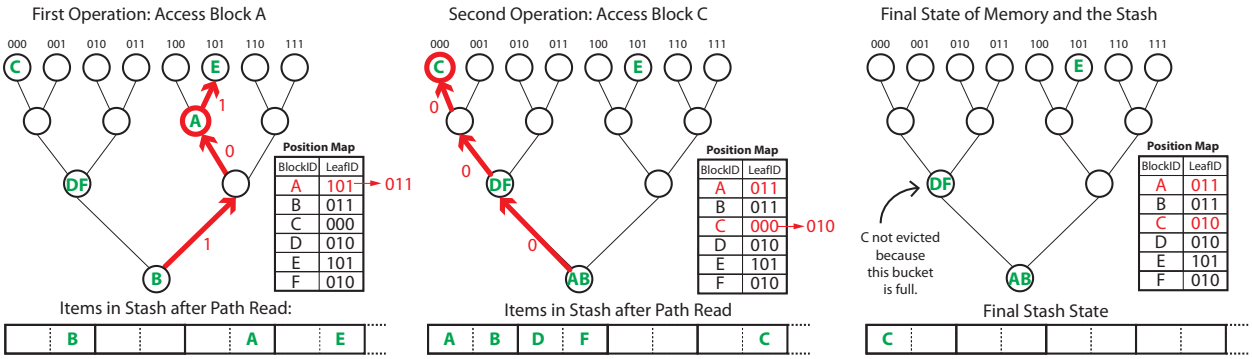


Figure 3.1: The Path ORAM Algorithm. The algorithm's operation is demonstrated for two path reads on a small ORAM tree. The first is a read of Block A, which the position map shows must be located somewhere on the path to leaf 101. Reading this path results in blocks B and E also being read into the stash. Block A is then randomly reassigned to leaf 011, and is therefore moved to the root of the path as it is being written back, since this is as far as it can now be inserted on the path to 101. Next, block C is read. Since the position map indicates that it is on the path to leaf bucket 000, that path is read, bringing blocks A, B, D, and F into the Stash as well. C is reassigned to leaf 010 and the bucket containing D and F is already full, so it can only be in the root of the path being written back. However, A and B must also be in the root as they cannot be moved any deeper, so C cannot be inserted. It therefore remains in the stash beyond the ORAM access.

block's designated leaf node. Reading this entire path into the *stash* – a data structure that stores data blocks in trusted memory – will thus necessarily retrieve the desired block along with other blocks on the path to the same leaf node.

After the requested block is found and its data returned to the requester (e.g., a CPU within a secure processor), Path ORAM reassigns the block to a random leaf node and then writes the same path back to memory that it had read before. Since the requested block was reassigned to a random leaf node, it may now belong to a different path from that on which it was read. As all paths emanate from the root, they will have at least the root node in common, but there is a 50% chance they will not share any others, in which case the reassigned block will have to stay in the root node. If no additional steps were taken, the upper levels of the tree would thus quickly become full. A Path ORAM implementation therefore has to move blocks in the stash as deep as possible towards the leaf of the current path as they are written back – this is called *reordering*. Furthermore, blocks may stay behind in the stash if there is no space for them in the path.

The obliviousness of Path ORAM stems from the fact that blocks are reassigned to random leaf nodes every time they are accessed. Repeated accesses to the same block will hence appear as accesses to a random sequence of paths through the tree (each of which consists of a full read followed by a full write of the same path). Algorithm 1 summarizes the Path ORAM algorithm, and Figure 3.1 illustrates its execution.

Algorithm 1 Pseudo-code of Path ORAM [83]

```
procedure ACCESS (block_id, read_write)
  if block_id in stash, access block there and exit
  leaf_id ← position_map [block_id]
  position_map [block_id] ← new random position
  path[] ← read path from root to leaf_id
  add all blocks found in path to stash
  if read_write = READ then
    return block with id block_id in stash
  else
    overwrite block with id block_id in stash
  end if
  write path from leaf_id to root (evicting as many blocks
    as possible and filling up with dummies)
```

Algorithmic Details

In addition to the basic algorithm, implementations of Path ORAM also need to take the following considerations into account:

- **Data Layout**: Path ORAM represents the full binary tree as a set of partitions in untrusted memory that each represent a node of the tree and are called *buckets*. Buckets are themselves divided into a fixed number of *slots* (usually four) that can each hold a single block and its associated *header*.
- **Encryption**: All data stored in untrusted memory has to be encrypted at all times, and is probabilistically reencrypted during every ORAM operation that touches it (otherwise it would be possible to correlate accesses to the same data). Each block's header therefore contains a nonce that changes every time the block is accessed, and affects the encryption of the entire block.
- **Dummies**: All slots of the tree that do not contain a block are filled with *dummies*, which contain no actual data but are encrypted in the same way as blocks so that their cipher text is indistinguishable from that of a block. Dummies are ignored for reordering and are not written into the stash.
- **Stash**: Even with reordering, there can be cases where not all blocks in the stash can be written back to the current path (Figure 3.1). This is addressed by making the stash larger than a path worth of blocks. Blocks that cannot be written back remain in the stash and are carried over into the next ORAM access and handled the same as if they had been read during that operation. At the start of an ORAM operation, it therefore has to be checked whether the block is in the stash already. If it is, a random path can be accessed to not leak this information.

Stash Overflows

It is important to note that the stash may *overflow* (i.e. no more blocks can be fit into the stash). Path ORAM can recover from overflows by reading and writing random paths and try to evict blocks from the stash during those path reads and writes. While this does not leak information (the random path accesses are indistinguishable from regular ORAM accesses), it increases execution time and may hence cause execution to not finish in the allotted time. It is therefore desirable to size the stash in such a way that these accesses occur rarely. In our CCS '13 paper [55], we presented an empirical analysis to determine a stash size that makes these overflows extremely unlikely. We summarize our findings from this work in Section 3.5.

Timing Channel

To avoid information leakage through memory access timing, Path ORAM can perform a non-stop sequence of path reads and writes, accessing a random path if there is no outstanding ORAM request from the CPU. Stash hits can be hidden by performing a fake path access as well, and multiple stash hits can be hidden behind the same access (alternatively, there can be a fixed gap between ORAM accesses, and stash hits can be performed within this gap). As described in Section 4.3, this is orthogonal to the microarchitectural details we investigate in Phantom (and hence not implemented), but would be required in a real deployment.

The timing channel has also been investigated by Fletcher et al. in a later project [22], and the results could be implemented in the context of Phantom as well.

3.5 Path ORAM Design Space

Path ORAM has a number of parameters that can be varied to result in ORAMs of different sizes and with different properties. The most important parameters are the *number of levels* in the binary tree and the ORAM *block size*. The number of levels in the tree determines the number of blocks that it can store and the size of the position map in trusted memory. In contrast, the block size allows to increase the size of the ORAM without affecting the size of the position map – however, a larger block size leads to a longer ORAM access latency and potentially more wasted data per memory access if the processor cannot use all data within a block. The exact trade-off depends on spatial (and, to a lesser degree, temporal) locality within program execution and is related to the trade-offs made when selecting the cache line size for a conventional processor.

While it is possible to vary the number of blocks per bucket and the number of blocks stored in the tree, we assume 4 blocks per bucket and store $4 \cdot 2^{l-1}$ blocks in a tree of l levels (i.e., as many blocks as there are in all leaf nodes combined). These values are chosen empirically and in line with the original Path ORAM paper [83].

Levels in ORAM tree	Size in no. of blocks	Size (128B blocks)	Size (4KB blocks)	Data per ORAM access
10	2^{11}	0.25 MB	8 MB	80x
13	2^{14}	2 MB	64 MB	104x
17	2^{18}	32 MB	1 GB	136x
21	2^{22}	512 MB	16 GB	168x
24	2^{25}	4 GB	128 GB	192x

Table 3.1: Overheads for different design points of Path ORAM. The table reports the design point, what ORAM size it would result in for different block sizes, and the amount of data movement per access relative to unprotected memory accesses (i.e., without ORAM).

Fundamental Overheads

Path ORAM always achieves perfect and provable obliviousness, albeit at both significant space and memory bandwidth costs. In particular, every memory access translates to an entire series of memory reads and writes, leading to an increase in data movement per memory access of more than 1-3 orders of magnitude, depending on the size of the ORAM and the block size. Furthermore, not the entire ORAM tree can be used to store ORAM content, leading to an overhead in usable memory as well.

With regard to space overheads, let the capacity N of the ORAM be defined as the number of logical blocks it can store. In the setting described above (i.e., we store as many blocks as there are slots in the leaf nodes), 50% of the physical memory is available as oblivious memory (which includes data and a 0.4% overhead for block headers). The remainder is reserved for dummy blocks.

The memory bandwidth overheads for different configurations are shown in Table 3.1. Since each block access results in an entire path read and write, Path ORAM's bandwidth overheads range from 104× for a 13-level ORAM tree (64MB capacity with 4KB blocks) to 192× for a 24-level tree (128GB capacity with a 4KB blocks). Consequently, the task of building a real, well-performing ORAM system becomes quite challenging: in particular, such a system will have to access two orders of magnitude more data per memory access.

Position Map Size

The majority of trusted storage is taken up by the position map. For an l-level tree, the position map has to store $(l-1) \cdot 2^{l+1}$ bits, i.e., $l-1$ bits for each ORAM block. As such, the position map size ranges from 2KB (for a 10-level tree) to 92MB for a 24-level tree. Naturally, some of these sizes are not acceptable in a secure processor scenario, as on-chip memories of more than 64MB are unlikely with current technology, even in the presence of more compact on-chip memories such as EDRAM [43].

Stash Size

In contrast to the position map, the size of the stash is flexible and has to be chosen by the designer. Smaller stash sizes make stash overflows more likely, which in turn results in longer execution times due to having to evict blocks from the stash until it frees up again. In this work, we focus on a scenario without such evictions and therefore choose the stash size such that overflows become extremely unlikely (we initially targeted an overflow probability of around 2^{-80} per access).

In [55], we show an empirical analysis based on long-running Path ORAM simulations for both SPEC benchmarks and synthetic worst-case workloads[3]. In these experiments, an infinite stash size was assumed and the number of entries remaining in the stash after every ORAM access was recorded. This allows to determine the fraction of ORAM accesses for which more than a certain stash size is required, which is equivalent to the overflow probability if that stash size is chosen.

While the simulation was run for billions of accesses, no simulation run could feasibly be long enough to determine the stash size such that the overflow probability becomes 2^{-80}. However, there exists a theoretical result [83] that relates the overflow probability to stash size and ORAM size N, up to constant factors. The results from the simulation runs were therefore used to fit these factors and derive the following formula for the minimum stash size to achieve an overflow probability of at most $2^{-\lambda}$:

$$2.19498 \log_2(N) + 1.56669\lambda - 10.98615$$

This minimum stash size includes two parts:

1. A $O(\log N)$ part for storing the path fetched from memory
2. A $O(\lambda)$ part for storing blocks left behind after the write-back phase

Based on the above formula, we designed the Phantom prototype with a stash size of 128 or 256 for $\log N = 13$ to 19, to achieve corresponding λ values from $\lambda = 70$ to $\lambda = 144$.

3.6 Path ORAM Optimizations

Numerous optimizations and extensions can be applied to Path ORAM. While we have not implemented them in the context of Phantom, they may become relevant for future work and expose additional microarchitectural challenges.

[3]These experiments and results are not considered a contribution of this thesis, since they were performed by Shi, Stefanov and Tiwari, inependently from the main author.

Hierarchical ORAM

As previously explained, the position map size can become very large for large ORAMs and/or small block sizes. If the position map becomes too large to be stored in trusted memory, it is possible to employ a hierarchical construction where the position map is stored in an ORAM itself and this ORAM is accessed in order to get the position map entry. This construction can be extended to an arbitrary number of levels to enable arbitrary ORAM and block sizes. However, the overhead of having to access multiple ORAMs for each access is significant and can easily outweigh the performance gain from smaller block sizes.

Ren et al. have investigated some of these trade-offs [69] and show optimizations for hierarchical ORAM that enable caching lower levels of the hierarchy through merging different levels of the hierarchy into the same ORAM tree [71].

Superblocks

Superblocks have been proposed as an additional optimization to Path ORAM [69]. In this optimization, adjacent blocks can be merged such that they are mapped to the same leaf node and are always brought into the stash together whenever one of them is accessed. This allows to exploit spatial locality in a program at the cost of additional stash overflows.

As with hierarchical ORAM, there is a trade-off between superblocks and choosing a larger block size. Both exploit spatial locality: superblocks do so at the cost of a larger position map (requiring more levels of hierarchy) while a larger block size increases the ORAM access latency. The exact nature of this trade-off is not agreed upon at this point and can only be determined in an Apples-to-Apples comparison on real applications spanning a large part of the design space.

Integrity & Freshness

In a real deployment, implementations of Path ORAM would require integrity and freshness in addition to confidentiality. The original Path ORAM tech report [81] discusses a Merkle-tree based approach to achieve this goal, which has later been expanded [70]. Since this is orthogonal to the problems investigated in Phantom, integrity and freshness were not implemented and are ignored for the remainder of the thesis.

Chapter 4

The PHANTOM Secure Processor

This chapter gives a high-level overview of the Phantom *Secure Processor and how we envision it to be deployed in a data center. It describes our attack model and introduces the different components of our design. Finally, it present a high-level overview of* Phantom*'s ORAM controller microarchitecture.*

4.1 Overview of PHANTOM

Phantom's basic setup is similar to traditional secure processors [88, 84]: it consists of trusted on-chip logic and memory whose activity cannot be observed externally. The key component of Phantom is its *oblivious memory controller* (or *ORAM controller*), which – from the processor's perspective – behaves like a traditional memory controller, but implements the Path ORAM algorithm to obfuscate all memory addresses that leave the chip and make the observable access pattern fully oblivious. In general, we envision Phantom to be integrated into a bigger system as a *coprocessor*.

We designed Phantom with today's data centers in mind – our goal was to introduce a design that could be deployed by cloud providers today and enables cloud customers to perform oblivious computation on encrypted data. One important aspect we considered was that such secure processors may be best deployed on FPGAs: considering the high costs of producing a custom ASIC and the fact that FPGAs are already starting to appear in data centers [18], an FPGA-based secure coprocessor is likely to be the most economical design point in the immediate future (unless the initial demand for obliviousness is sufficiently high to justify an ASIC implementation).

While we focus on FPGAs for the rest of this thesis (and have prototyped Phantom on an FPGA-platform), our microarchitecture could be implemented within a custom ASIC as well, and even be deployed in novel architecture designs such as the Hybrid Memory Cube [62]. As such, we believe that the results we learned while building Phantom are universally applicable to any implementation of a Path-ORAM-based oblivious secure processor.

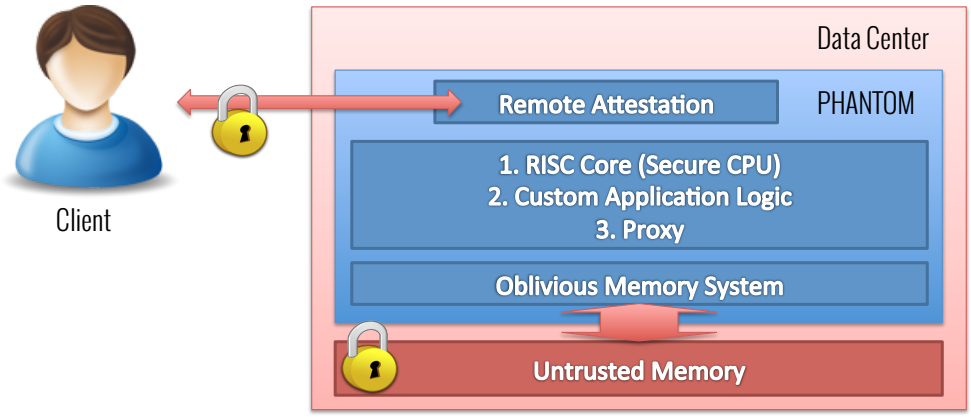


Figure 4.1: The different possible usage scenarios for PHANTOM. The ORAM controller can either be used with a Secure CPU, a custom domain-specific accelerator or forward requests from a remote client.

4.2 Usage Model

We consider a general setting where a *user* wants to offload sensitive data and computation to the cloud, a *cloud provider* sells infrastructure as a service (IaaS), and a *hardware manufacturer* creates secure FPGAs. We envision the cloud provider to deploy a *secure processor* on a secure FPGA bought from the hardware manufacturer. This secure processor, in collaboration with the secure FPGA, provides a mechanism for remote attestation to the user. Through this mechanisms, the user can ensure that she is running computation on a genuine secure processor, without having to trust the cloud provider.

We envision the secure processor to have non-volatile memory on-chip to store a unique private key (such FPGAs are available off-the-shelf [24] and are a direct replacement for the FPGAs on our prototype platform). A remote client can establish a session key with a *loader* program on the secure processor and then transfer an encrypted ORAM image with sensitive data and code to the processor's physical memory (i.e., DRAM). The loader then executes the code obliviously and stores results back into ORAM. The remote client can collect the encrypted results once the time allocated for the computation has elapsed.

PHANTOM is our take at this secure processor. We envision it to be deployed in three different usage scenarios (Figure 4.1):

1. The remote client stores all sensitive computation, including both code and data, into PHANTOM's ORAM. PHANTOM, which comprises a general-purpose RISC core and an ORAM controller on a trusted chip (an FPGA in our prototype), then executes this program obliviously.

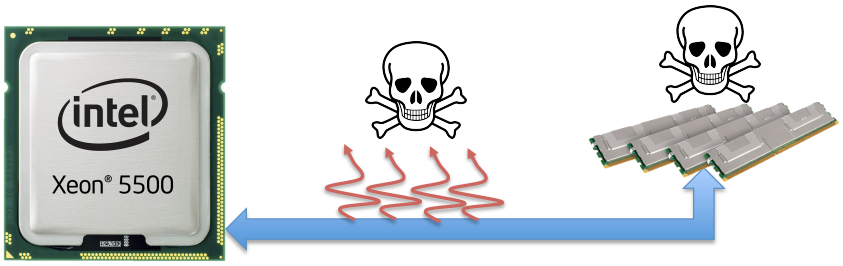


Figure 4.2: Visualization of the attack model

2. In cases where high performance is required, the RISC core can be replaced by custom logic that implements the computation directly, similar to existing FPGA-based domain-specific accelerators [98, 75]. These accelerators directly connect to the ORAM controller, so that access patterns remain protected. As such, PHANTOM can allow an existing hardware accelerator to become oblivious on-demand, by using the ORAM controller instead of accessing memory directly.

3. The remote user runs a trusted program on a standard processor (after establishing a dynamic root of trust), and does her best to keep sensitive data in confidential on-chip memory [66]. When sensitive data must be spilled off-chip, the trusted program makes encrypted ORAM reads/writes through PHANTOM.

The third scenario is harder to make verifiably secure because preventing plain-text data from going off-chip on a commercial microprocessor is complicated. However, it has the benefit that existing applications can be ported easily and can run faster than on the FPGA. This approach could also be used for a (less efficient) software implementation of ORAM.

4.3 Attack Model

We aim to protect against untrusted cloud providers and third parties with physical access to the cloud provider's data center. We trust users and hardware manufacturers (malicious hardware attacks [90] are out of scope) and focus on digital attacks where an attacker with physical access to the machine can probe memory busses, board-level interconnects and chip pins (Figure 4.2). Such attacks can originate from a number of different sources:

- **Malicious employees**: Admins in a data center require full physical access to machines to perform their duty, and this privilege can be abused. As recent leaks at the National Security Agency [35] have shown, not even requiring top secret security clearance for all admins is sufficient to prevent – potentially devastating – leaks. It is also conceivable that malicious employees are sent to infiltrate data centers. Other examples of malicious employee attackers includes disgruntled employees out for revenge [19].

- **Data center break-ins**: For high-profile targets such as banks, government organizations or targets of industrial espionage, sophisticated break-ins into data centers are in the scope of economic viability. A case study [74] has shown that even moderate effort sufficed to infiltrate a (purportedly) high-security data center and gain physical access to machines (despite physical measures such as cameras and security desks).

- **Government surveillance**: With large amounts of potentially intelligence-critical data available in the cloud, they make a very attractive target for governments. As recent security leaks have shown, physical attacks involving hardware implants for server machines are in active use by the intelligence community [25] and it can be presumed that these are not limited to the United States. Similarly, when offloading workloads to a third country, users have no guarantee that the cloud provider does not have to comply with government-mandated surveillance, since the existence of such surveillance measures is often kept secret and cloud providers are not allowed to disclose them to their users. This has been of great concern to customers and is expected to have a significant impact on cloud provider revenues [12].

Once an attacker has physical access to a machine, one way she could probe the memory interconnect would be by installing malicious DRAM DIMMs that contain non-volatile memory in addition to DRAM (similar DRAM modules exist for the purpose of backing up data in the case of a power outage [4, 58]). These DIMMs could log (or sample) accessed addresses to this non-volatile memory, and the attacker could later extract and analyze them.

While existing solutions (such as Intel SGX [57] and secure processor such as XOM [88] or AEGIS [84]) only consider explicit data leakage through the data bus (and prevent it with encryption), we extend this attack model to *implicit* information leakage through the address bus. In addition to encryption, we therefore require full memory trace obliviousness (Section 3) to consider a system secure for our purposes.

Note that the *total* execution time (a termination channel) is out of scope for this work. However, information leaks through this channel can be countered by computing the worst case execution time for a program, or through offline program analysis to set execution times that are independent of confidential data. Probabilistic solutions to this problem are acceptable, since a computation that does not finish by the desired execution time can be cancelled and return an error to the user (which is invisible to an attacker).

Further, the timing of *individual* ORAM accesses does not leak information if Phantom is deployed such that a non-stop stream of DRAM traffic is maintained (whether or not there are outstanding ORAM requests – if there are not, the ORAM controller can perform fake accesses to random paths). Cache hits inside the CPU or the ORAM controller would not alter the pattern of DRAM accesses observable by an adversary and only reduce the execution time (i.e. timing channels are reduced to a termination channel).

We do not consider software-level digital attacks where malicious software relies on covert channels through the processor or operating system. Such attacks can be addressed using architectural and OS support for strong isolation [89, 44], obfuscation [56], and deterministic

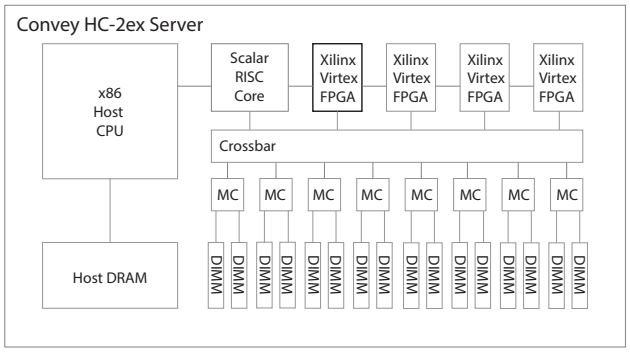


Figure 4.3: The Convey HC-2ex Heterogeneous Computing Platform

execution [5, 17]. Analog attacks that exploit the physical side-effects of computation – such as temperature, EM radiation, or even power draw – are orthogonal to our proposal as well. These can be addressed through techniques that normalize or randomize the potential physical side-effects of computation. Further, timing and termination channels can be eliminated by normalizing the program execution time and rate of memory accesses by the secure CPU. Alternate obfuscation approaches [56] exist as well.

4.4 Implementation Platform

We prototyped Phantom on a Convey HC-2ex server [15]. The HC2-ex is a heterogeneous computing platform with a server-grade Intel Xeon CPU connected to a custom board that features four large FPGAs (Xilinx Virtex-6 LX760) connected in a ring, and 16 independent memory channels, each with 64 DRAM banks, for a combined memory of 64GB (Figure 4.3). Between the four FPGAs, this high-bandwidth memory system can achieve a bandwidth of 80GB/s (20GB/s per FPGA) and is also coherent with the host memory. In addition to these main components, a small RISC core next to the FPGAs manages communication between the host CPU and the FPGAs, which becomes as simple as calling into a special function on the host that then triggers functionality on the FPGAs (Section 6.2).

Phantom is implemented as an independent co-processor on one of the HC-2ex's FPGAs. Oblivious computation is initiated by the host (by setting up the code and data to operate on), after which Phantom executes the code obliviously and stores the result back to memory. We also use the host CPU for debugging purposes (e.g. for debug output).

This choice of platform was based on the insight that Path ORAM's main bottleneck is its large memory bandwidth overhead – a high-bandwidth memory system such as the one available on the HC-2ex is therefore an opportunity to make ORAM practical by providing the required memory bandwidth within off-the-shelf hardware. Furthermore, the reconfigurable logic available on the Convey system both gave us a convenient prototyping platform and a realistic deployment scenario (the HC-2ex is commercially available).

While we started with a set of preliminary simulations to understand the trade-offs in the Path ORAM algorithm and the implementation challenges we may face, we quickly moved to prototyping the ORAM controller at the RTL-level.

Creating a full RTL-implementation was necessary to learn about the details that need to be considered in a real Path ORAM controller. Many of them are not apparent from the high-level description of the algorithm, such as the importance of how to reorder blocks, how to organize the stash, how to arrange data in DRAM, what meta-data needs to be stored with each block and when to update it (e.g., we found that blocks should store their own leaf ID to decrease pressure on the position map, and that it needs to be updated during the read, not the write phase). Hence, without building a full hardware-implementation, it is not clear what to simulate, since nobody had ever built an ORAM system before.

The decision to not merely simulate at the RTL-level but go all the way to a full FPGA-based hardware implementation was based on three considerations:

- Many issues are not visible at the RTL-level, in particular issues related to meeting timing constraints. When building Phantom, we found many places where logic that seemed innocent at the algorithmic and RTL level caused overly long wires when synthesized to real hardware, and required us to adapt the design. For example, our heap-sorter design (Section 5.6) is motivated by such issues. If simulating a system only at a high level, one cannot be confident that such issues won't arise and invalidate the numbers.

- We wanted to simulate our ORAM controller with real-world workloads running for a long time. This is necessary to confirm that it behaves as expected in a real deployment, and also increases confidence in its correctness. However, simulating a single ORAM access in Synopsys VCS took 30 seconds for our ORAM controller – not enough to simulate more than a few hundred accesses. Running on an FPGA allowed us to run more than 30,000 ORAM accesses per second and execute real-world workloads such as SQLite. Notably, there were a number of subtle bugs in the ORAM controller that only manifested after 10,000 accesses or more – without running on real hardware, we would have never found those bugs. As such, building the real system significantly helps to increase confidence in our results as well.

- We wanted to demonstrate a prototype that – with moderate changes –could be deployed on hardware that exists today. This would allow other researchers to use Phantom as a prototyping platform for hardware-related ORAM research, and may also help to drive commercial designs should they arise.

The Convey HC-2ex enabled us to achieve all of these goals. However, very little of the Phantom implementation is specific to this particular platform, and it would be possible to port Phantom to other platforms as well, or use it as an IP block in an ASIC.

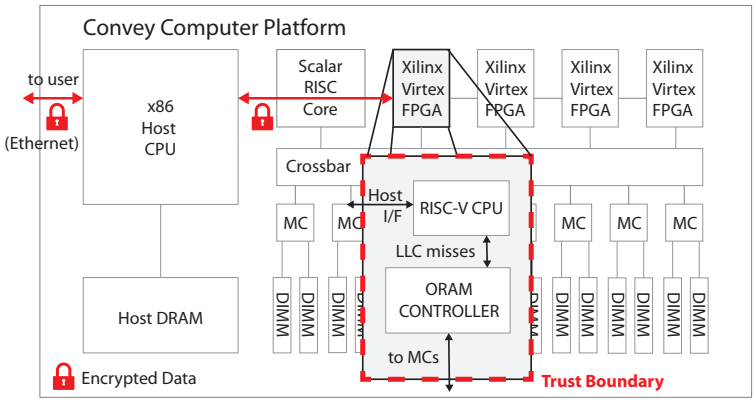


Figure 4.4: High-level System Design. PHANTOM comprises a CPU, non-volatile memory, and an ORAM controller implemented on a single FPGA. All *digital* signals outside the PHANTOM chip are assumed to be visible to an adversary.

FPGAs: Opportunities & Challenges

FPGAs give us an opportunity to prototype a processor with an oblivious memory system under realistic conditions. However, designing high-performance logic on an FPGA is challenging. FPGAs operate at much lower frequencies than custom chips – for instance, most of the FPGAs on the Convey platform are clocked at 150MHz – because logic is implemented as a network of interconnected look-up tables.

The main challenge of PHANTOM is therefore to map Path ORAM, an irregular data-driven algorithm, onto slow FPGA logic and yet ensure that 1) PHANTOM maximizes memory bandwidth utilization, and 2) execution time of an ORAM access is independent of the access pattern (since it would otherwise leak information). Sections 4.6 and 4.7 describe how we achieve these two requirements.

4.5 System Design

PHANTOM consists of our custom designed ORAM controller and a single-core in-order RISC-V [87] CPU developed in Berkeley's computer architecture group. The entire project was implemented in Chisel [8], a new hardware description language developed at UC Berkeley. PHANTOM is implemented on one of the Convey HC-2ex's four FPGAs (Figure 4.4), but some design points use up to two more FPGAs to store the position map.

The ORAM controller is connected to the processor's memory interface, in place of a conventional memory controller. As such, it handles all LLC misses from the CPU – a thin conversion layer translates the CPU's request format to the ORAM controller's interface and converts between the (large) ORAM block size and the (small) cache line size of the CPU. The ORAM controller provides an efficient microarchitecture implementing the Path

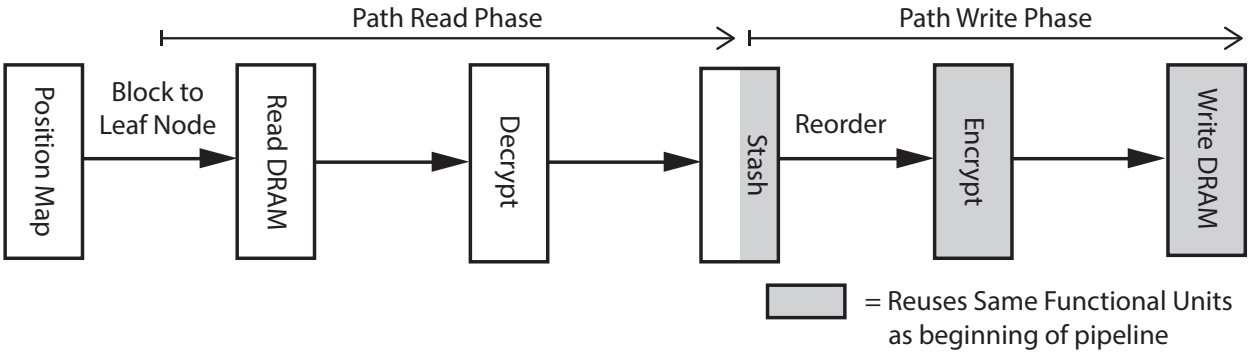


Figure 4.5: Overview of the steps of the Path ORAM algorithm.

ORAM algorithm, which makes use of Convey's high-bandwidth memory interface and uses a number of technique to keep up with the memory system.

The RISC-V CPU communicates with the host-processor through a bidirectional communication channel, the *host-target interface* (Section 6.1). This allows the host to initialize the program to execute obliviously and provide debugging facilities to the RISC-V CPU. It also allows the host to bypass the RISC-V CPU to write to oblivious memory directly.

Note that everything outside the FPGA is assumed to be untrusted, in accordance with our attack model (Section 4.3). This includes the channels between the host CPU and the FPGA, as well as between the ORAM controller and the memory controllers. As a result, any communication on these channels has to be encrypted[1].

The ORAM Controller

Whenever the ORAM controller receives a request to access an address in oblivious memory (usually on a last-level cache miss from the CPU), it initiates an ORAM access. It first looks up the desired ORAM block in the position map to get a leaf node to access, then reads and decrypts the path to that leaf and puts the result into the stash. It can then return the data in the accessed block (or update it in case of a write) before encrypting and writing back the entire path. While doing so, it has to reorder the blocks on the path to move them as far down the tree as possible. Figure 4.5 presents an overview of this process and shows the corresponding components of the ORAM controller.

The following sections give a high-level overview of the ORAM controller and Chapter 5 will talk in detail about its microarchitecture. Chapter 6 will then describe the integration of the ORAM controller with the RISC-V processor and the HC-2ex.

[1]In our prototype, we do not implement encryption, but model it as fixed-length pipelines where required.

Levels (N)	17	19	17
Block size	4096	4096	1024
Stash size O(C)	128	256	128
1 MC (baseline)	34816	38912	8704
8 MCs, pick from stash	10880	21888	9248
8 MCs, non-overlapped sort	5248	6912	1984
8 MCs, overlapped sort (ours)	4352	4864	1088

Table 4.1: Potential improvement from optimizations in cycles (or time) per ORAM access. Baseline is an ORAM processor with one memory controller (MC). PHANTOM uses 8 MCs and adapts Path ORAM's reordering step; we expect it to be 8× better than the baseline.

4.6 Achieving High Performance

Like all ORAM schemes, Path ORAM has a fundamental overhead in the amount of data that needs to be accessed per memory request, e.g. 136× for a 1GB ORAM with 4KB block size (Section 3.5). Path ORAM's bandwidth requirement thus motivates the use of platforms with a very wide memory system, such as the Convey HC-2ex: by employing a 1,024b-wide memory system rather than a conventional 128b-wide one, we can achieve a potential speed-up of 8× over a naive implementation on a traditional memory system. However, exploiting the higher memory bandwidth is non-trivial: it is necessary to co-optimize the hardware implementation and the Path ORAM algorithm itself, in order for the implementation of Path ORAM to keep up with the memory system.

We now present the key ideas and optimizations that allow us to achieve this goal. Table 4.1 summarizes the potential performance gains of each idea over a naive implementation with a traditional memory system. These numbers are based on back-of-the-envelope calculations and are called 'potential' because they assume a perfect PHANTOM implementation without DRAM stalls. We demonstrate experimentally in Chapter 7 that PHANTOM comes close to this ideal on the actual FPGA implementation.

Table 4.1 presents three different ORAM configurations and a baseline design that uses a standard, 128b-wide memory controller (MC) operating at 150MHz. The baseline assumes that all ORAM logic is free and accesses are limited only by the bandwidth of a single MC, namely 34,816 cycles to access a 17-level ORAM with 4KB blocks. Our potential performance-gains are therefore conservative estimates.

Memory Layout to Improve Utilization

Simply using a wide memory bus does not yield maximum DRAM utilization. Concurrent accesses to addresses in the same bank – bank conflicts – lead to stalls and decrease DRAM bandwidth utlilization. To resolve such bank conflicts, we chose a layout of the Path ORAM tree in memory (DRAM) where data is striped across memory banks, ensuring that *all*

DRAM controllers can return a value almost every cycle following an initial setup phase. Fully using eight memory controllers in parallel thus reduces the 136× bandwidth overhead to 17×, making ORAM controller logic on the FPGA the main implementation bottleneck.

Picking Blocks for Writeback

Bringing 1,024b per cycle into Phantom raises acute performance problems: the operation of the Path ORAM algorithm now has to complete much faster than it did before, to keep up with the memory system (e.g., Phantom needs to decrypt and encrypt 1,024 bits per cycle in parallel). While encryption can be parallelized by using multiple AES units in counter mode, the Path ORAM algorithm still has to manage its stash and decide which blocks from the stash to write back after each ORAM access (the reordering step from Section 3.4).

The latter is of particular importance: The ORAM controller has to find the blocks that can be placed deepest into the current path, and do so while the memory controllers push in and write out data at 1,024b per cycle. One approach would be to scan the entire stash and pick a possible block for every position on the path, in parallel with writing the path to memory in leaf-to-root order. However, with Convey's high memory bandwidth, scanning through the stash takes longer than writing out an ORAM block, causing this approach to achieve less than half the potential performance with stash size $C = 128$ (Table 4.1).

At the same time, in order to keep the stash small, it is crucial to select each block from the *entire* stash – including both the current path blocks and old blocks. In the CCS paper [55], we show that an approach that only considers the top blocks of the stash would not suffice and could cause unbounded stash growth.

We hence propose an approach that splits the task of picking the blocks for writeback into two phases: a *sorting phase* that sorts blocks by how far they can be moved down the current path (XORing their leaf with the leaf ID of the current path), and a *selection stage* that (during bottom-up writeback) looks at the last block in the sorted list and checks in one cycle whether it can be written into the current position – if not, no other block can, and we have to write a dummy (Section 5.6).

We further improve on this approach by replacing the sorting and selection stages by a min-heap (sorted by the current path's leaf ID). This replaces an $O(C \log C)$ operation by multiple $O(\log C)$ operations (each of which completely overlaps either with a block arriving from or being written to memory), where C is the size of the stash. This now makes it possible to overlap sorting completely with the path read and to overlap selecting with the path write phase, allowing the Path ORAM logic to keep up with the memory system (Table 4.1). Section 5.6 describes this approach in more detail.

Treetop Caching inside the Stash

While the stash is required by Path ORAM as a temporary store for ORAM blocks while they wait to be written out, it can also be used to improve performance by securely caching ORAM blocks on-chip. We propose to cache the top levels of the ORAM tree inside the stash

– we call this *treetop caching* – which saves PHANTOM from fetching these parts of the path from DRAM. Since the number of nodes is low at levels close to the root, caching a few levels improves ORAM latency and throughput significantly while using only modest amounts of trusted memory. Our results demonstrate this insight experimentally (Figure 7.1).

We designed PHANTOM's stash management to support treetop caching with minimal effort (as well as other methods, such as LRU caching). To do so, we use a content-addressable memory (CAM) that serves as lookup-table for entries in the stash, but is also used as directory for caching and as free-list to find empty slots in the stash. This avoids placing caches separate from the stash – one of our previous prototypes showed that this leads to performance delays from checking the cache and moving ORAM blocks between the cache and the stash (and additionally complicates PHANTOM's logic, which makes obliviousness harder to ensure).

Meeting FPGA Constraints

Each BRAM on our FPGAs is limited to 2 read/write ports. However, the BRAMs in PHANTOM that constitute the stash have to be multiplexed among four functional units (encryption during path reads, decryption during path writes and reads/writes by the secure CPU). We therefore designed PHANTOM such that all the units that read from or write to the stash are carefully scheduled such that only a single read port and a single write port on the BRAM is in use during any particular clock cycle. Implementing the min-heap to reorder stash entries also requires similar tricks, which we describe in Section 5.6.

The FPGAs also impose strict timing constraints on our design. Convey's DRAM controllers operate at 150MHz, which means that the ORAM controller had to run at 150 Mhz as well. Since this frequency was too high for the CPU (which was originally designed for ASICs), we put it into a second 75 Mhz clock domain available on the Convey architecture, and use asynchronous FIFOs to connect it to the ORAM controller. Nonetheless, without rearchitecting part of the uncore (Section 6.1), we are only able to run with cache sizes of 4KB (IC), 4KB (DC) and 8KB (LLC), while still overclocking the circuit slightly (synthesis results differ between design points, but a representative example has 73 and 142 Mhz for the respective clock domains). To extrapolate numbers for realistic cache sizes, we therefore ran simulations based on the ORAM access latencies from our prototype (Section 7.5).

To run at these frequencies, we had to modify a baseline PHANTOM implementation to replicate and pipeline critical paths. The details can be found in Chapter 5.

4.7 Preserving Security

When implementing a complex security-critical algorithm such as Path ORAM, it is crucial to ensure that the hardware implementation does not introduce any additional leaks, e.g. through timing channels. One way to achieve this goal would have been to use formal verification – however, verifying large hardware designs is a very resource-intensive process

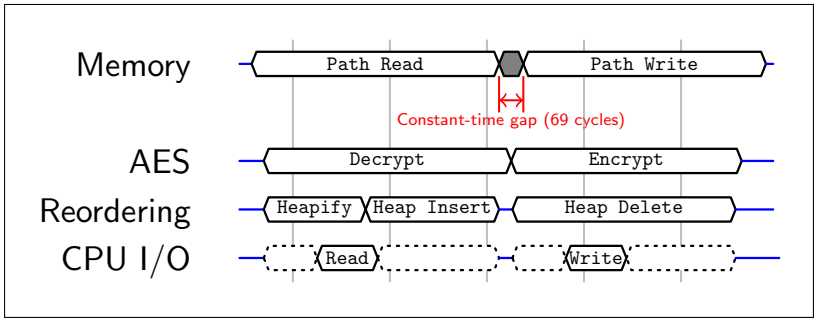


Figure 4.6: Overlapping the steps of the Path ORAM algorithm with memory accesses

that is hard to achieve in an academic setting. We therefore chose a different approach and designed PHANTOM such that it is free of information leakage by design. To achieve this goal, we rely on the following two techniques: 1) we use a set of design principles to avoid timing variations in PHANTOM's operations, and 2) we isolate interactions with the outside world from PHANTOM's internal (confidential) data structures.

Design Principles for Obliviousness

We use two simple design principles to ensure that PHANTOM's design does not break Path ORAM's obliviousness guarantees. Any operation – checking the position map, reordering, caching, etc. – that depends on ORAM data is either a) statically fixed to take the worst-case time or b) is overlapped with another operation that takes strictly longer. PHANTOM's decrypt operation could, for example, be optimized by not decrypting dummy ORAM blocks – but this leaks information since it would cause an operation to finish earlier depending on whether the last block was a dummy or not. Instead, PHANTOM pushes dummy blocks through the decryption units just the same as actual data blocks. These two design principles yield a completely deterministic PHANTOM pipeline. Figure 4.6 shows how the different operations in PHANTOM overlap with reading and writing a path.

Terminating Timing Channels at the Periphery

The DRAM interface requires further attention to ensure security. PHANTOM sends path addresses to all DRAM controllers in parallel, but these controllers do not always return values in sync with each other. Although DRAM stalls do not compromise obliviousness (DRAM activity is not confidential), propagating these timing variations into PHANTOM's design can make PHANTOM's timing analysis complicated. To keep PHANTOM's internal behavior deterministic and simple to analyze, we introduce DRAM buffers at the interface with external DRAMs to isolate the rest of PHANTOM's ORAM controller from timing variations in the memory system. At the same time, all inputs to the DRAM interface and their timing are public (a leaf ID and 1,024b of encrypted data per cycle during writeback), so that no information can be leaked out of PHANTOM.

Chapter 5

Microarchitectural Details

This chapter presents the details of the microarchitecture of Phantom*'s ORAM Controller. It discusses design decisions and challenges, and highlights aspects of implementing Path ORAM in hardware that would not have been apparent without a real implementation.*

5.1 High-level Description

Since the fundamental limitation of Path ORAM is its memory bandwidth consumption, the goal of Phantom's microarchitecture is to make maximum use of the available DRAM bandwidth (up to 1,024b/cycle on the Convey HC-2ex). This involves accelerating all computation related to Path ORAM such that it can be moved off the critical path and occur in parallel with the memory accesses, as far as possible. An optimal implementation would be one that performs no additional cycles beyond those needed for the memory accesses – as we will see in this chapter, Phantom gets close to this goal.

Figure 5.1 shows how the different components of the microarchitecture fit together and Figure 5.2 illustrates Phantom's data path in further detail. First, ORAM blocks are read from memory, decrypted, and written to the stash – we call this the *read phase*. After the read phase, blocks are read from the stash, encrypted, and written back to memory using the *same functional units* as in the read phase; this is the *write phase*. Concurrently to the read and write phases, Phantom reorders the blocks in the stash and returns a value to the secure CPU (or writes a value into an ORAM block).

5.2 On-Chip Data Structures

The *Position Map* (❶ in Figure 5.2) is the central data structure for making memory accesses oblivious, and stores a mapping from ORAM blocks to the leaf node in the ORAM tree that the block is assigned to.

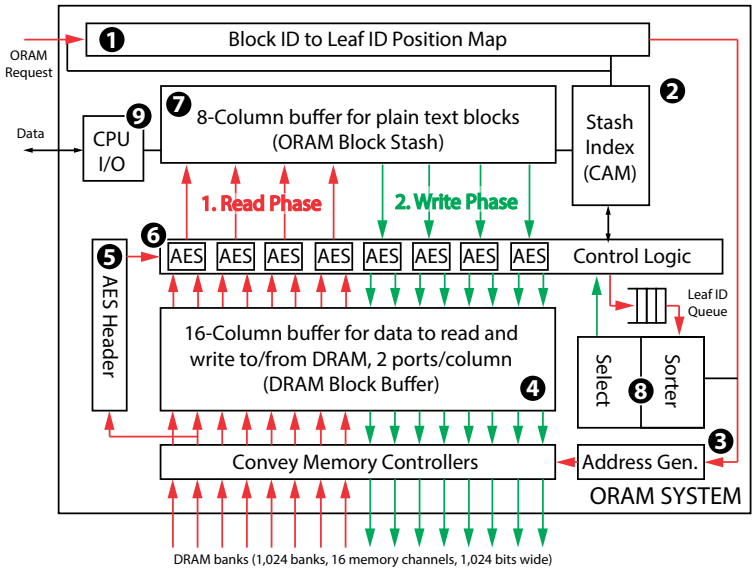


Figure 5.1: Overview of the ORAM controller. The arrows represent the data flow during the two phases of the Path ORAM algorithm.

The *Stash* ❼ buffers ORAM blocks in plaintext. Its size is chosen to make the overflow probability small (Section 3.5). Any block that is decrypted in the read phase is put into the stash, and blocks are removed from the stash when they are encrypted in the write phase. All blocks stored in the stash are striped across 8 independent *columns* (i.e., memories), each accessed by an AES unit (Section 5.5). This is necessary to reduce the critical path (and achieve the clock frequencies required for the Convey platform).

The *Stash Index* ❷ is a content-addressable memory (CAM) that stores the address of the block contained in each entry of the stash, including a bit to indicate whether a stash entry is occupied. Before decrypting a block, the stash index is queried to find a free slot. During a path write, once encryption of a block finishes, the corresponding entry of the Stash Index is set to free.

The Stash Index is queried before starting each ORAM access, in parallel with the position map. If the requested block is present, the ORAM controller will read or write to the stash directly, rather than initiating a Path ORAM access (as described in Section 3.4, in a real deployment, PHANTOM would have to perform a fake path access to hide the fact that the stash was accessed and avoid a timing channel – our prototype does not implement this at the time of writing).

Querying the Stash Index and Position Map adds a single cycle latency to each request. As the Position Map grows linearly with ORAM size, it may also have to be pipelined to keep the critical timing path low (up to 4-deep for the position map of a 17-level tree), which adds another few cycles of latency to each request (4 in the example above). Updating the stash index requires 2 cycles, but these are overlapped with the path reads/writes.

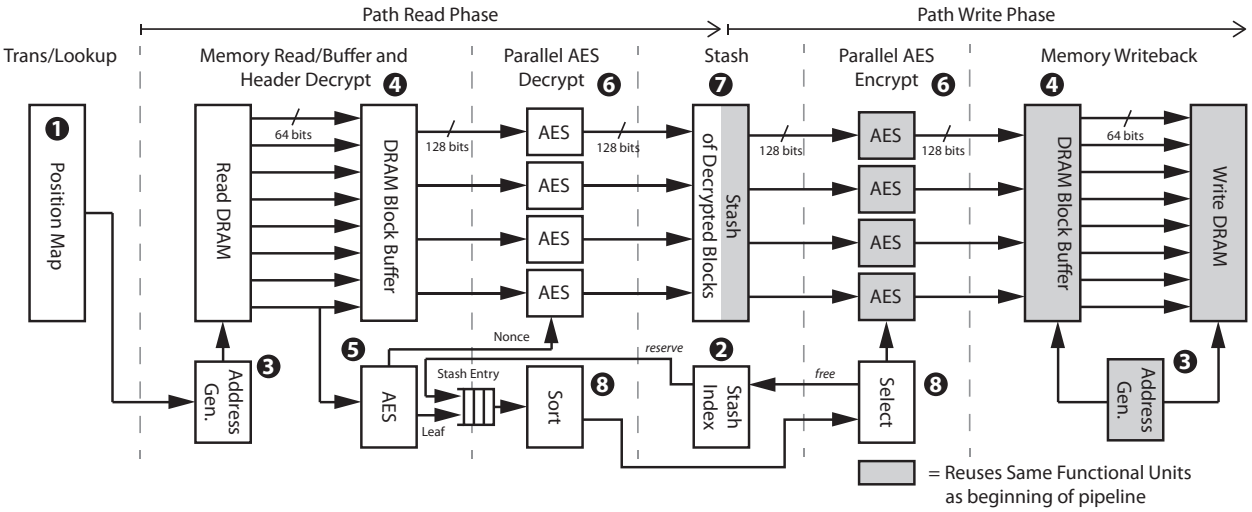


Figure 5.2: The data path in PHANTOM's ORAM controller: Data is read, decrypted and written to the stash. It is then read from the stash, encrypted and written back to memory, using the same functional units.

5.3 In-Memory Data Structures

The main data structure in DRAM is the ORAM tree (Section 3.4). It consists of a number of *buckets*, each of which stores 4 *ORAM blocks* of size 4KB. Each ORAM block then consists of a 128b header (for meta-data) and its payload. Since Path ORAM's performance is dominated by memory accesses to this data structure, its memory layout is crucial.

The Convey platform features 1,024 DRAM banks (64 for each of its 16 memory channels). During an ORAM access, the *address generation logic* ❸ takes the leaf ID of the path to access and generates a series of memory requests for each of these channels. Our goal is to fetch data at (close to) peak bandwidth – to do so, however, we cannot place the words of the ORAM blocks sequentially in the memory address space. Instead, we need to place each word such that the DRAM banks' latency can be hidden behind successive 64-bit words being read from different DRAM banks (or, alternatively, bursts of words).

To achieve this goal, we stripe each ORAM bucket across the 1,024 banks such that, when reading a block sequentially, every memory controller accesses all its banks in sequence, reading a 64-bit word from each bank and wrapping around after the last bank. Going round-robin across all DRAM banks allows each bank enough time to open a DRAM row and return its data to the memory controller without the controller having to stall. Our implementation uses the fact that some bits in Convey's memory addresses directly map to certain DRAM banks – we can hence ensure that successive words map to different banks.

The layout of the tree itself follows the standard way of storing a binary tree in memory: The root is stored at address $bucket\_size/16$ (for each of the 16 memory channels), and the two children of the bucket at address i are located at $2 \times i$ and $2 \times i + bucket\_size/16$.

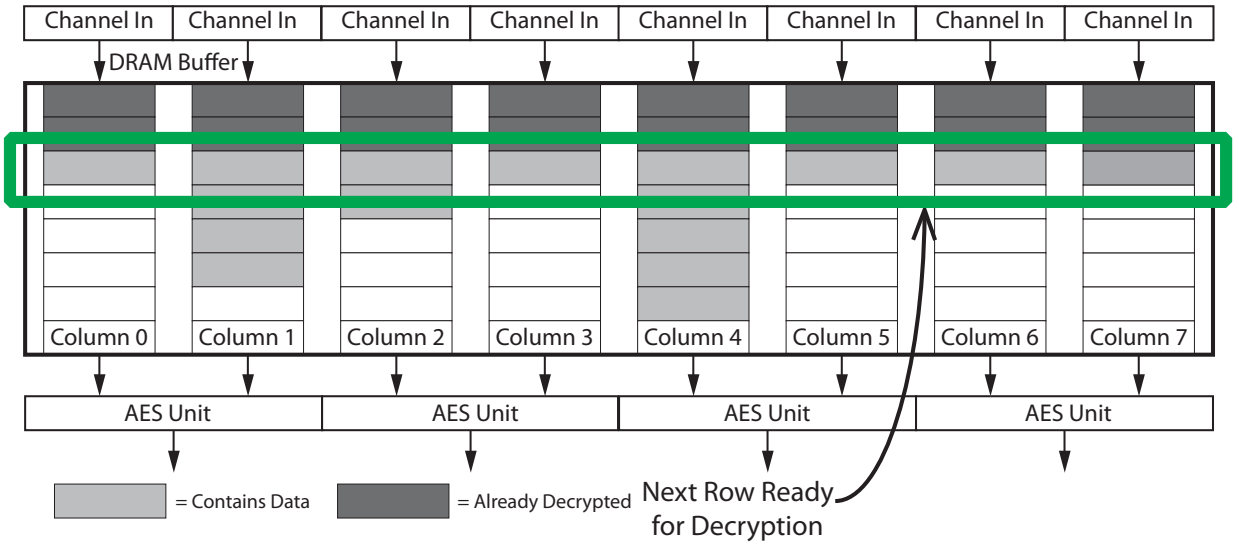


Figure 5.3: The DRAM Buffer. Data is written as it returns from the memory controllers, and fed into the AES units at a rate of 1,024b/cycle as soon as an entire row is ready.

5.4 DRAM Buffer

During an ORAM access, the address-generation logic generates memory requests for each word in the path. When encrypted data arrives from memory, it is put into the *DRAM Buffer* ❹ in order to be decrypted in subsequent stages. In practice, DRAM controllers often run ahead or fall behind, and the DRAM Buffer waits until it receives a complete row – the maximum number of bits received from DRAM each cycle (1,024 bits on the HC2-ex) – before forwarding it into the AES decryption units. To account for worst-case DRAM stalls, we provision the DRAM Buffer with space to store an entire path of ORAM blocks. During the write phase, the DRAM Buffer stores encrypted data from the AES units on its way to be written back to memory, absorbing DRAM stalls without stalling AES encryption.

The DRAM Buffer's design allows us to isolate the internals of Phantom from timing variations in the memory system. As long as the core part of the ORAM controller (i.e. encryption/decryption hardware, stash and reordering logic) consumes 1,024 bits of data per cycle out of the DRAM buffer, and produces data at this rate during write-back, Phantom is completely shielded from any DRAM timing variations (Figure 5.4).

The DRAM Buffer is implemented using 16 independent columns (Figure 5.3). Each column serves one memory channel and keeps track of how much data it has received from the memory system – we use a feature of the HC-2ex that forces the memory controllers to buffer responses internally and deliver them in order, so that the DRAM Buffer only has to store how many words each column has received. As soon as all the words of a row are available, the buffer notifies the AES units that the row is ready to be consumed.

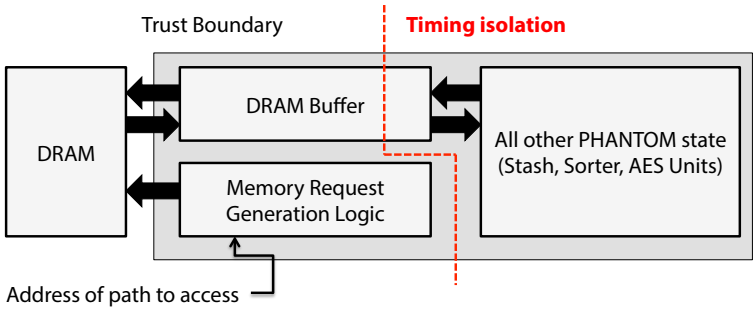


Figure 5.4: The DRAM Buffer protects PHANTOM's internal state from timing variations in the memory system. As such, it prevents timing channels by design.

5.5 Encryption of ORAM Blocks

Once data has arrived in the DRAM buffer, it is consumed by AES units to be decrypted into the stash. The same units also encrypt all data as it is written back from the stash into the DRAM buffer. PHANTOM uses eight AES-128 units in counter mode (CTR). This approach was chosen since counter mode allows us to parallelize both encryption and decryption, which is crucial to maintain the required throughput of 1,024 bits per cycle.

Each ORAM block has an associated clear-text nonce for CTR (which could be stored in its first 128 bits). It also includes an (encrypted) block header which includes the block's ID and whether or not it is a dummy. As an optimization, we store the block's leaf ID in the header as well – this reduces accesses to the position map.

Before the AES units can start decryption, they first need to have the block's nonce available to them. To avoid this causing stalls during the read phase, it is possible to add a forwarding path ❺ from the first two memory channels. The forwarding path buffers all words belonging to nonces in a way similar to the DRAM buffer, so that the nonces are available when the AES units start to decrypt the actual block.

To avoid wasting oblivious storage, we introduce an additional optimization: we store the nonce in the block header itself (since more than 80 out of 128 header bits are free – even for large configurations[1]). The nonce now gets encrypted, since it is included in the header AES block. Thus, the forwarding path must be extended with an AES unit (❺) that pre-decrypts the header while the rest of the block is being fetched, and (due to overlapping and in-order memory fetch) is always finished by the time the rest of the block is available to be decrypted. Since this makes the AES unit's timing completely deterministic based on the timing within the memory system, it does not break any of the security guarantees.

Decrypting the header in advance also allows PHANTOM to insert a block into the reorder min-heap (Section 5.6) while the rest of the block is still being read. PHANTOM also reserves

[1]The highest-order bit of the non-nonce part should be set to 1, to avoid the counter of AES-CTR ever coinciding with that part, as the resulting ciphertext would be identical to the encrypted header.

an entry in the stash once the header is decrypted; subsequent outputs of the AES decrypt units can then be directly written to this entry without any stalls, thereby hiding the 2-cycle stall of the stash index lookup (Section 5.2).

The choice of AES implementation is a trade-off that is largely orthogonal to our architecture, as long as each AES unit is pipelined and provides 128 bit/cycle throughput. In our prototype, we model AES-128 units as 11-stage pipelines that take the nonce, 128b plain text and counter as input, and output the cipher text 11 cycles later. We also assume an on-chip pseudo-random number generator (PRNG)[2]. While we do not include the size of AES and PRNG in our results, we show that PHANTOM's Oblivious RAM controller requires less than 2% of the logic on the FPGA, leaving ample space for AES and PRNG functionality (e.g. 9 instances of the AES unit from [39] would require 50% of the available LUTs).

5.6 Heap-based Reordering

As described in Section 3.4, the blocks in the stash have to be reordered before being written back to memory, ensuring that each block is moved as far down the path as possible. A simple implementation would be to go through the entire stash for each slot, in order to choose a candidate block. However, this search would take a larger number of cycles than the write-back of the block to memory would take, which would make the ORAM access computationally bound rather than memory bound (especially on an FPGA, where clock rates are much lower than on an ASIC).

To avoid this problem, PHANTOM has $max\_time = (blocksize/bits\_per\_cycle)$ cycles to pick a block to write back. In our prototype with its 4KB block size, *max_time* is 32. With a stash of about 100 entries, a linear scan will not be hidden behind a 32-cycle ORAM block read, and will thus introduce a 68-cycle stall per block or ($68 \times path\_length \times blocks\_per\_bucket$) cycles for each ORAM access – 4,624 additional cycles for a 17-level ORAM. To avoid these stalls, we need a different approach that can overlap reordering with the memory accesses.

It is important to note that in some cases, the linear scan approach may take fewer than 100 cycles – e.g. if the stash contains fewer than 100 elements. However, to avoid information leakage through a timing channel, the scanning operation always needs to take the maximum time possible, even if it could finish earlier (which follows from our design principles for avoiding timing channels from Section 4.7). Any alternative approach we choose therefore has to make every decision within *max_time*, even in the worst case.

An alternative to scanning the entire stash is to pre-sort its entries based on their leaf IDs. PHANTOM reorders stash entries by performing a bit-wise XOR between the leaf IDs of entries in the stash and the leaf ID of the current path (assuming the leaf ID naming scheme from Figure 3.1), and then *sorting* (❽) them in increasing order. The resulting order reflects how deeply they can be placed in the tree, since the first bit where they disagree with the current leaf ID is at a later, lower-order bit if they can be placed deeper.

[2]Instead of generating truly random numbers, our prototype uses a linear feedback shift register to emulate the PRNG functionality (which is insecure but sufficient for prototyping purposes).

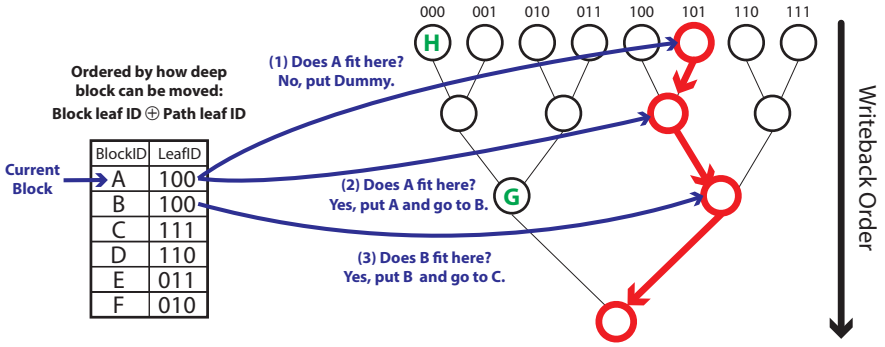


Figure 5.5: Reordering approach based on sorting the blocks on the path and considering the first unplaced block for each slot. Each bucket stores one block in this example.

Once entries have been sorted like this, PHANTOM determines which blocks to write back by going backwards through the path (starting from the leaf), always looking at the lowest element in the sorter and checking its leaf ID to determine whether it can be placed in the currently examined slot in the tree. If yes, PHANTOM inserts the block, otherwise PHANTOM fills the slot with a dummy (dummies are generated directly by the AES units – instead of reading data from the stash, they simply encrypt some random data). We call this the *select* stage (❽). Figure 5.5 visualizes this strategy.

While this approach avoids going through the entire stash, the sorting step requires further attention. Algorithms like Merge sort cannot overlap the sorting with the memory reads and writes (because they require all entries to be present at the time the sort begins). On the other hand, algorithms like Insertion sort (which is simple to implement in hardware and can perform the insertion in parallel to the reads) would make the latency to insert each block linear in the stash size and larger than *max_time* (i.e., 32 cycles).

We therefore designed PHANTOM to use a min-heap data structure instead. Rather than performing all the sorting either during the write phase or during the read phase, we insert the blocks into the min-heap as they are read in, and remove them from the heap as we write them back. This takes logarithmic time per block both during the read and write phases, but as a result, the phases can individually keep up with the speed at which blocks arrive from memory and have to be written back.

We also have to make sure that we *heapify* (re-sort) the current contents of the heap at the start of *each* ORAM operation, since the ordering changes with the leaf ID of the current path. We therefore added a queue (which may hold as many entries as the stash) to buffer the meta-data of incoming blocks in case the heapify operation is not complete by the time the first blocks come out of the AES unit. We can show that the overall latency does not exceed the amount of time we have during the read phase, ensuring that we always finish all insertions into the heap by the time the last block has finished reading (a proof follows).

With these optimizations, we can overlap the reordering completely with the memory requests, making our system completely limited by the available memory bandwidth.

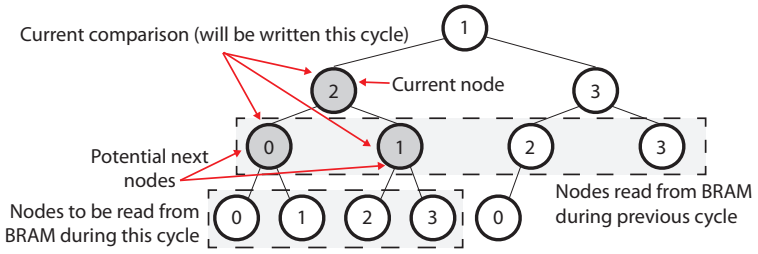


Figure 5.6: Min-Heap implementation using the FPGA's BRAMs. The numbers represent the BRAM each node is stored in. The four second-degree children of each node are put into four separate BRAMs, so that they can be prefetched in parallel.

Implementing the Min-Heap on an FPGA

The heap implementation requires some care to minimize the cost of each operation. For some heap operations, a node must be compared to both of its children, potentially doubling the access latency to the heap (from $\lceil \log k \rceil$ cycles to $2\lceil \log k \rceil$ cycles, where k is the number of nodes in the heap), since each of the FPGA's BRAM memories has only one read and one write port. We avoided this problem by splitting the heap into two separate memories, each with a read and a write port. One holds even memory locations, the other holds the odd memory locations. As a result, the two children of a node are always in different BRAMs and can be accessed in the same cycle. The updates to the heap are performed in parallel to those reads, using the remaining write port.

Although this approach is functionally correct, it results in a circuit with paths from one BRAM through multiple levels of logic to another BRAM, leading to a long critical path latency. We therefore split the memory even further: our final implementation uses 4 different BRAMs (Figure 5.6). At every cycle, we prefetch the four grandchildren of a node so that the two children we are interested in will be available in buffer registers at the point when they are required for comparison (whichever way the previous comparison went).

Proof of Overlapping Heapify

For the heap operations to fully overlap with the memory reads and writes, it is necessary to ensure that the heapify (re-sort) operation at the start of each ORAM access will always complete in time as well. Both the heapify operation and inserting all the incoming blocks into the heap must perfectly overlap with the memory read phase for this to be the case. We show that for realistically sized ORAMs (at least 13 levels, stash size at most 256), this is always the case and the heap-based approach is correct.

Proof. For a heap with tree-depth d, *insert* and *extractMin* take $d + 3$ cycles. For a stash size less than 256, $d \leq 8$ and hence the *extractMin*s overlap with the (32 cycle) writes since

$8 + 3 \leq 32$. To prove that the insertions overlap as well, we have to show that the *heapify* operation and all insertions always finish in the available time.

Heapify starts at the first cycle of the ORAM access, and incoming blocks from from AES decrypt are stored in a queue. Hence it is sufficient to show that the combined time for *heapify* and all *inserts* is smaller than the time the entire path read takes. Since *inserts* take $d + 3$ cycles per block, for an ORAM tree with l levels and a bucket size of 4, we need to ensure that *heapify* takes at most $t_{max} = 4l(32 - (d + 3))$ cycles.

Heapify performs a $k + 1$ cycle operation for each node in the heap except the ones on the last level, where k is the distance of the node to the leaves. Hence *heapify* takes

$$t_h = \sum_{i=1}^{d-1} 2^{i-1}(d - i + 1) < \sum_{i=1}^{d-1} 2^{i-1}2^{d-i} = \sum_{i=1}^{d-1} 2^{d-1} = (d - 1)2^{d-1}$$

cycles. Now if we set $d \leq 8$, $l \geq 13$, then $t_h < 7 \cdot 2^7 = 896$ while $t_{max} = 4 \cdot 13 \cdot (32 - (8 + 3)) = 1,092$. Hence $t_h < t_{max}$ and therefore *heapify* and *inserts* overlap. Note that this bound is not very tight (for $d \leq 8$, $t_h = 374$) and sorting can be further optimized. □

5.7 Control Logic

Phantom's control logic is the component that coordinates the movement of data through the ORAM controller, namely from the DRAM buffer through the AES units into the stash (and vice versa for the write-back), maintaining the invariant that data takes constant number of cycles to pass through each stage of the pipeline.

At its core, the control logic consists of a pipeline that runs in parallel to the pipelined AES units. Each stage of the pipeline holds a *descriptor* for the data in the corresponding stage of the AES units (Figure 5.7). The descriptor is a tuple comprising an ORAM header (i.e. block ID, leaf ID, whether it is a dummy or not and a nonce), an entry in the DRAM buffer (to read from), and an entry in the stash (to write to). An offset determines which row within the DRAM buffer and stash the descriptor's data belongs to.

During the read phase, the control logic reads rows of data from the appropriate location in the DRAM buffer and feeds them into the AES units. It also directs the output of the AES units into the appropriate locations in the stash. In addition, the control logic checks each decrypted block header to see if it is the block the CPU has requested. Once the block is found, it is returned to the CPU and also remapped to a new, random leaf node in the ORAM tree (overwriting the block's header before writing it into the stash).

If the ORAM access is a read, the control logic forwards the requested block to the CPU while it comes out of the AES units. The forwarding avoids the problem that each of the BRAMs comprising the stash only has one read and one write port. If we allowed the CPU to read the block from the stash at any time, it might clash with reads during the write-back. By returning the data to the CPU before it is written to the stash, we avoid this problem.

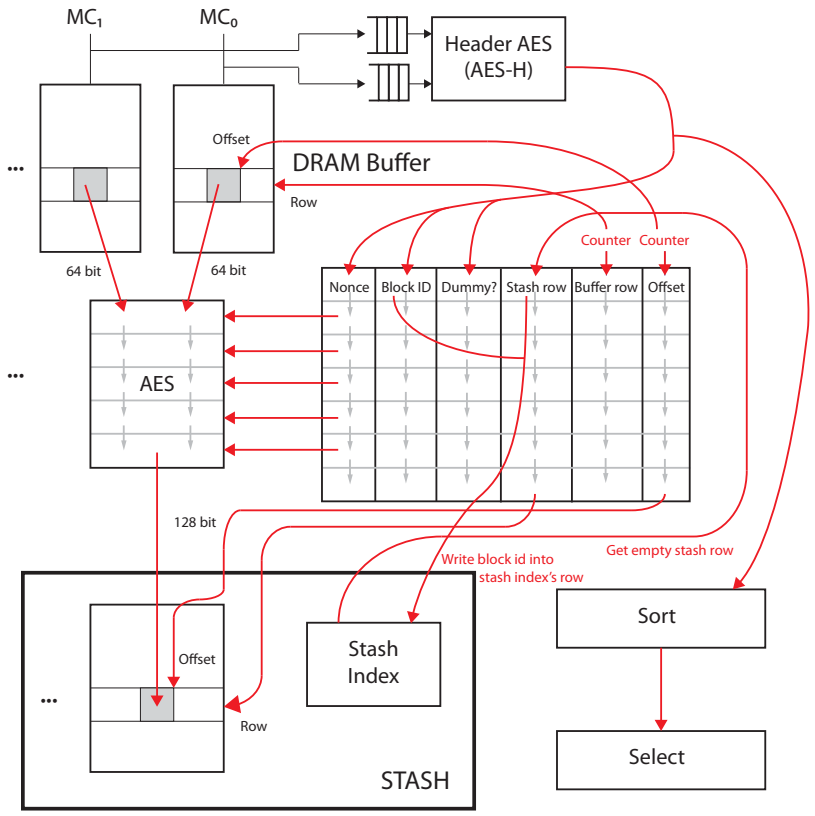


Figure 5.7: PHANTOM's control logic. The arrows describe the flow of data.

The write phase uses the same pipeline and AES units, but with different descriptors. As data is moved out of the stash to the DRAM buffer, the control logic queries the select stage (Section 5.6) to determine whether the next block to be written is a dummy, and if not, which entry from the stash to write back. During this step, all block headers are also updated with a new nonce generated by the PRNG.

If the ORAM access is a write, the CPU's block will be used to update the ORAM block in the stash in parallel to the write-back phase, so that the updated ORAM block is available for write-back at least one cycle ahead of the encryption unit – this is necessary since the stash's write port is only available during the write-back phase. In the case of a dummy, no data is read from the stash and zeros are fed to the unit instead (since the nonce is randomly generated and affects the whole block, zero-valued dummies do not affect security).

The remainder of the write phase resembles the read phase: data is written to the DRAM Buffer in order from bottom to the top of the path and is guaranteed to arrive at 1,024 bits per cycle, so that the memory system can write at full pin bandwidth (1,024 bits/cycle).

5.8 Treetop Caching

An advantage of this design is that it enables us to implement a number of optimizations very easily. To implement our Treetop Caching optimization (Section 4.6) that caches the top levels of the tree, we only need to modify the control logic and memory request generation logic to only read the bottom levels of the tree (which is a trivial change of a few lines of code). As a result, any blocks that would be stored in the top levels of the tree remains behind in the stash after an access, and will be in the stash already when an access begins.

It would be similarly easy to implement other optimizations such as LRU caching in the stash. In that case, the min-heap logic could be modified to give a sorting value of ∞ to each entry that should remain in the stash according to the LRU logic. As a result, these entries will be the last to arrive at the select stage and can be ignored when writing the path back to memory.

5.9 Utilizing Multiple FPGAs

The Convey HC-2ex features 4 FPGAs that provide additional logic and memory capacity which can be used for ORAM state. For example, we experimented with splitting the position map across the FPGAs adjacent to the one carrying the ORAM controller (since the FPGAs on the HC-2ex are connected in a ring, each can directly signal both of its neighbors). This allows us to scale to larger ORAM sizes.

When performing a position map access (which only happens at the start of an ORAM request), we send the (encrypted) address of the block we are looking for, as well as the new leaf ID to map it to, to both of the neighboring FPGAs. Both of them then send an encrypted reply at the same time. Since it is public knowledge that an ORAM request was initiated, this is secure.

Chapter 6

Implementation on the HC-2ex

This Chapter describes how the ORAM controller is integrated with the RISC-V CPU and how the resulting system is deployed on the Convey HC-2ex platform. It also shows details of the Convey HC-2ex's programming interface and how it is used to implement Phantom. *The remainder of our infrastructure, including the use of the Chisel hardware description language and our experiences with both Chisel and the HC-2ex, are covered as well.*

6.1 Integration with a RISC-V Processor

As described in Section 4.5, Phantom consists of our ORAM controller integrated with an in-order RISC processor implementing the RISC-V instruction set [92]. The processor (like our ORAM controller) is implemented using Chisel [8], a new hardware construction language embedded in Scala (Section 6.4). The CPU features a full implementation of the RISC-V instruction set, an *uncore* that communicates with the memory system and a *host*, as well as instruction, data and last-level caches (LLC). For our prototype, we use a version

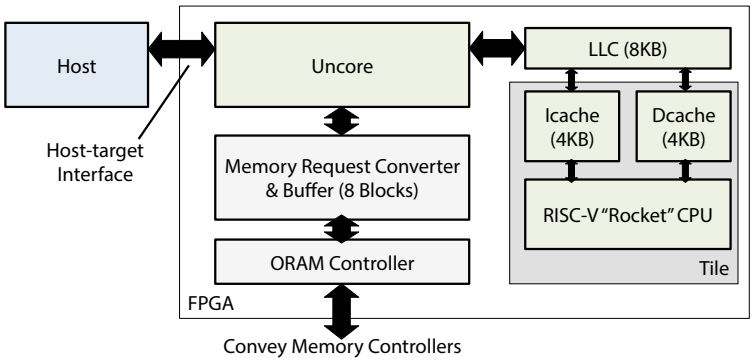


Figure 6.1: Integration of the ORAM Controller with the RISC-V CPU. Components in green are part of the RISC-V infrastructure while components in grey are part of Phantom.

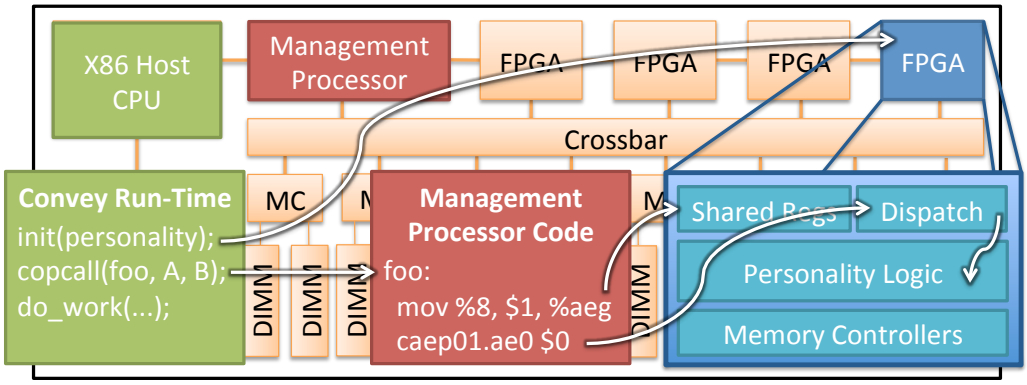


Figure 6.2: Overview of the Convey HC-2ex platform.

with very small caches (4KB instruction cache, 4KB data cache, 8KB LLC) and no floating-point unit. The reason for this choice is that our Convey platform requires the CPU to run at 75 MHz on an FPGA, and supporting larger cache sizes at this frequency would have caused significant amounts of additional work (the CPU was optimized for an ASIC implementation before we adopted it for Phantom and running at 75 MHz still slightly overclocks the circuit). Such work is orthogonal to our proposed ORAM controller. For our performance evaluation, we therefore simulate a CPU with realistic cache sizes to determine results for a real deployment.

The CPU is integrated with Phantom through a *Memory Request Converter* unit that translates memory requests from the LLC into ORAM requests, buffering 8 previously accessed ORAM blocks to handle the difference between 128B cache lines and 4KB ORAM blocks (Figure 6.1). We use the CLOCK algorithm [86] to determine which ORAM block to evict from the buffer when it fills up.

To translate from the CPU's memory requests to ORAM requests, it is necessary to divide the memory request's address by the exact ORAM block size. Since 128b of the block are used for the block header, this requires division by a non-power of two (an alternative approach would be to store the headers separate from the payload). The memory request converter in our current prototype uses a Xilinx divider IP core that takes an additional 36 cycles, but custom logic could exploit the fact that this division is by a power-of-two multiple of 31, to reduce that latency to just a few cycles.

The CPU communicates with the host (in our case, the host CPU on the Convey HC-2ex) via the *host-target-interface* (HTIF), a bidirectional communication channel. The software that controls the RISC-V CPU is taken from the standard RISC-V infrastructure (which is available for download from [87]). Specifically, the host CPU runs a *frontend-server* which is responsible for loading an executable, initializing the RISC-V CPU through the HTIF and handling any requests that the RISC-V CPU is sending through the HTIF. The RISC-V processor itself runs a minimal *proxy kernel* that enables basic syscalls by forwarding them

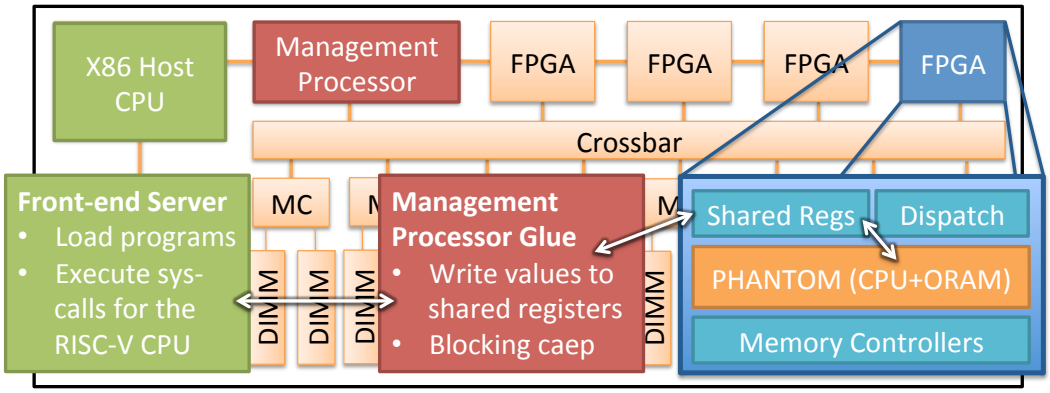


Figure 6.3: Building a communication channel between the host CPU and PHANTOM.

to the frontend-server on the host. While this would break security in PHANTOM (and is therefore not used by our sample applications such as SQLite), we found this feature helpful for debugging and timing measurements.

6.2 PHANTOM on the Convey HC-2ex

Convey provides a development kit for the HC-2ex which includes infrastructure IP and a set of scripts for the Xilinx design tools that synthesize and package FPGA images into a *personality* (a package that is automatically loaded by the Convey runtime to reprogram the FPGAs for a particular purpose). Personality-specific logic is implemented in a Verilog module that interfaces with the rest of the system (such as the memory controllers), and the host CPU communicates with personalities through a *management processor* (MP) that is connected to the FPGAs and runs code supplied by the host. The MP can access shared registers within each personality and has special instructions that are directly dispatched to personalities running on the FPGA. The Convey compiler bundles MP code (written in assembly) with the host binary, and host applications can perform so called *copcalls* ("co-processor calls") to run a function on the MP. This workflow is shown in Figure 6.2.

PHANTOM is implemented as a personality and uses these mechanisms to implement the host-target interface between the host CPU and the RISC-V core on the FPGA (Section 6.1). We provide copcalls that access a set of shared registers, which we connect to the host-target interface of the RISC-V *uncore*. This allows us to exchange commands with the RISC-V system (such as accessing control registers or writing to memory).

Figure 6.3 shows how the different parts fit together: The front-end server is running on the host and takes care of tasks such as loading programs. It will then send commands over the HTIF by calling into copcalls that transfer the HTIF data into registers on the FPGA, which are then fed into the RISC-V processor's uncore.

6.3 Debugging Support

During the development of Phantom, we built a significant amount of debugging functionality to debug the ORAM controller both in simulation and on the FPGA. Convey provides an extensive debugging infrastructure for the HC-2ex, and we used this infrastructure with Synopsys VCS. We make extensive use of the fact that host and Convey memory are coherent: for example, we implemented a software version of the Path ORAM algorithm and use it as a reference to compare against the state of the tree between ORAM accesses, to confirm the correctness of our hardware implementation. In addition, we use this implementation to initialize the ORAM tree for testing purposes.

We also added a number of debug ports to Phantom, for example to write to the position map from the host. Finally, we implemented a visualization tool that shows us the current state of the ORAM tree after every access, including the location of blocks.

6.4 Chisel as an Implementation Language

Phantom was almost entirely developed in Chisel [8] (except for some glue code to integrate it with the Convey infrastructure). Chisel is a hardware description language embedded in Scala. While it describes synthesizable circuits directly (similar to Verilog), it makes the full Scala programming language available for circuit generation, enabling functional or recursive descriptions of circuits. Chisel also has additional features compared to Verilog, such as width inference and a type system for wires, support for structs (*Bundles* in Chisel terminology), high-level description of state machines and bulk wiring.

Chisel supports a number of backends that enable it to (for example) generate a fast RTL-level simulator implemented in C++, which is significantly faster than VCS (we used this feature to test our min-heap implementation). Chisel can also generate synthesizable Verilog code, which made it easy to interface with the Convey infrastructure.

To give an intuition about the clear coding style that Chisel enables, Figure 6.4 shows the implementation of the insert step in the min-heap implementation. During this operation, an entry is first added to the end of the min-heap tree. At every step, it is compared with its parent: if it is larger or equal, the insert operation terminates, if the parent is larger, the two are swapped and the insert operation proceeds at the next level.

One complication in our design is the fact that the heap is divided into four different memories to reduce the critical path (Section 5.6). In Verilog, it would be difficult to determine the right memory to read each node from. Chisel, however, makes this easy by enabling us to implement functions `read_buf` and `write_buf` that take a node address and transparently read or write to the correct memory. Similarly, `cur_buf` enables to extract a node's value from the set of nodes that were read in the last cycle.

Another interesting aspect to Chisel is how it hides the notion of clock cycles: instead of operating at clock edges, it has `when` and `otherwise` statements that check whether a statement is true during the current cycle and updates state for the next cycle.

```
when (state === s_insert) {
    // The current parent has been read last cycle, so we can extract the value.
    val parent_entry = cur_buf(next_ptr);

    // Read the next parent so it is ready in the next cycle
    read_buf(next_ptr >> UInt(1));

    when (last_ptr === UInt(1)) {
      // Root has been reached
      write_buf(last_ptr, next_entry);
      state := s_delay;
    }.elsewhen (compare_bits(parent_entry) > compare_bits(next_entry)) {
      // The parent is larger and we need to swap them and continue
      write_buf(last_ptr, parent_entry);
      next_ptr := next_ptr >> UInt(1);
      last_ptr := next_ptr;
    }.otherwise {
      // The parent is smaller and we can finish
      write_buf(last_ptr, next_entry);
      state := s_delay;
    }
  }
```

Figure 6.4: An example of Chisel code as it appears in PHANTOM. This code snippet implements the insertion operation for the min-heap from Section 5.6. `last_ptr` is the location of the current node in the tree, `next_ptr` is the location of its parent, `next_entry` is the value that is being inserted into the heap and `parent_entry` is the value of the parent.

As a result, the insert implementation becomes comparatively simple: it distinguishes three cases (the root has been reached, the parent is larger than the current node, or otherwise). In the second case, the operation continues one level up in the tree, writing the parent's value into the current node location. The other two cases terminate the operation.

6.5 Experiences with the Infrastructure

Since we were using two pieces of infrastructure that are not very widely used at this point – specifically the Convey HC-2ex and the Chisel Hardware Implementation language – we want to make a few comments regarding their usefulness with regard to the research we were conducting. Overall, we believe that an academic project of the size of PHANTOM would have been significantly harder had it not been for the availability of these high-quality tools.

Convey HC-2ex

We found the HC-2ex to be well-suited for ORAM research due to the simple interfacing of host and FPGA. Further, the Convey memory system offers many opportunities for experiments, e.g. with parallel requests or recursive ORAM.

Overall, the HC-2ex gave us most of the features that we needed for our research. However, one useful additional feature would have been the ability to handle cache misses from the host CPU on the FPGA, since this would enable us to run the program on the host CPU and send encrypted memory requests to an ORAM controller on the FPGA. However, there appears to be no fundamental reason this could not be provided by the hardware in the future, if there is demand for this feature.

Chisel

Chisel significantly helped the implementation of Phantom, since it made it much easier to generate parts of the design automatically based on design parameters, and deal with complex control logic (in particular due to interlocking in different parts of the pipeline). As such, the implementation effort for Phantom would have been much higher had we not been able to use Chisel.

6.6 Real-world Deployment

Phantom is a prototype, and would therefore require a number of additional changes to be deployed in a security-critical cloud scenario. The most important omission of our prototype are facilities for remote attestation, which would have to be provided in a real Phantom deployment. Furthermore, our current prototype does not implement real AES units – these could be implemented with commercially available IP blocks and Phantom leaves enough remaining logic blocks on the FPGA for this to be possible. Other features that are currently missing are integrity and freshness, as well as handling of the timing channel.

In addition to these microarchitectural features, a real deployment would require the Convey HC-2ex's FPGAs to be replaced by secure FPGAs instead [24]. Furthermore, some more work would have to be done on the software side to enable time-sharing of Phantom and forward requests between remote clients and the secure processor.

Overall, we believe that Phantom answers the questions we set out to answer – how to implement an oblivious processor in real hardware and what microarchitectural challenges need to be overcome. However, as with any prototype, more work is needed to make it practically available and we believe that our results could help to guide this work.

Chapter 7

Evaluation

This Chapter presents our evaluation of Phantom *to determine the hardware and performance overheads of obliviousness, as well as its performance impact on real applications. It reports numbers from our hardware implementation for both individual ORAM accesses and SQLite workloads. In addition, we use a timing model and simulator to extrapolate results for different cache sizes.*

7.1 Evaluation Highlights

In this section, we experimentally determine the cost of obliviousness on Phantom, as well as its performance impact on real applications. Our evaluation demonstrates that:

1. ORAM latency to access a 4KB block is 32× over a non-secure access (44× for 128B blocks[1]) for a 1GB ORAM. ORAM access latencies vary from 18us to 30us for ORAMs of effective size 64MB to 4GB and a block size of 4KB. In comparison, a non-ORAM access takes 812ns for a 4KB block and 592ns for a 128B block when using all of the Convey HC-2ex's memory channels.

2. Phantom utilizes 93% of the theoretical peak bandwidth of Convey's memory system for one FPGA, validating our memory layout and microarchitectural optimizations.

3. Phantom's ORAM-controller requires less than 2% of the LUTs of the FPGA, and varying amounts of BRAMs. Phantom's prototype implements ORAMs of up to 1GB on a single FPGA (before on-chip memory is exhausted), and 2GB or larger by splitting the position map across multiple FPGAs (Section 5.9).

4. Overall application performance depends on access patterns and working-set size. As an example, we ran different SQLite queries on a population census database. The overheads over a non-oblivious version ranged from 20% to 500%.

[1]Since the granularity of 4KB ORAM accesses is much larger than a cache line, we compare to 128B reads to estimate the worst-case overhead, which occurs if only one cache line of the data is actually used.

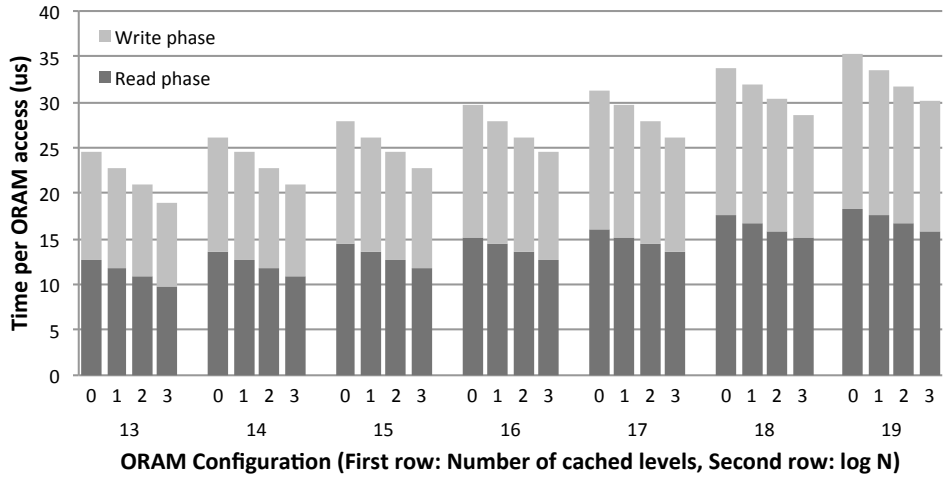


Figure 7.1: Average time per ORAM access for different ORAM configurations. The numbers were gathered on the real hardware, performing 1M accesses per experiment with block size 4KB. The 18 and 19 level design points are split across 3 FPGAs.

7.2 ORAM Latency and Bandwidth

The primary metric of efficiency for an ORAM system is the time to service a single request to the ORAM. This approximates the amount of overhead per memory access (i.e. last-level cache miss). While this number says little about the performance impact on real applications, it allows us to put an upper bound on the expected overheads.

We synthesized a set of different configurations of Phantom, with effective ORAM storage capacity ranging from 64MB to 4GB (13-19 tree levels with a 4KB block size). Each ORAM configuration includes a stash of 128 elements, which allows up to 3 levels of Treetop Caching. For 18 and 19-level ORAMs, Phantom's position map is stored on one and two adjacent FPGAs respectively (Section 5.9). Note that none of these designs include support for integrity and freshness, which would be required in a real deployment.

Figure 7.1 shows the total time per ORAM access for these configurations. The experiments were conducted on the FPGAs of our Convey HC-2ex machine. We performed a sequence of 1 million random ORAM accesses (i.e. block reads and writes) for each experiment, and report the average times per access (note that times for accesses may vary without compromising security, but only due to timing variations in the DRAM system).

For each data point, we also report how long it takes until the data is available – this is important for reads, since a CPU using the memory system can continue execution as soon as the data is returned from ORAM. For writes, execution can continue immediately.

We measured that Phantom's latency until ORAM data is available ranges from 10us for a 13-level (64MB) ORAM to 16us for a 19-level (4GB) ORAM. Compared to sequentially reading a 4KB (812ns) or a 128B (592ns) block of data using all Convey memory controllers,

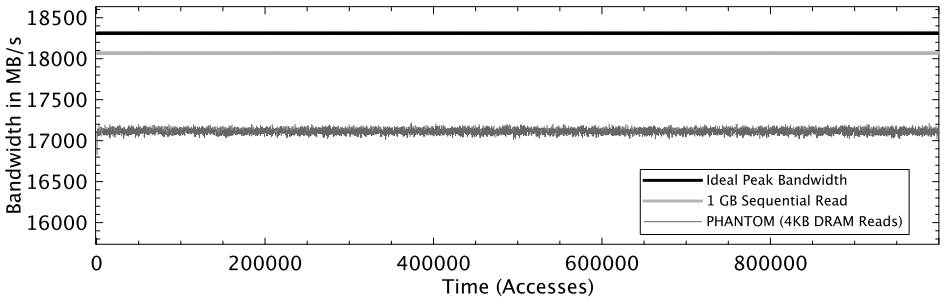


Figure 7.2: DRAM bandwidth utilization over time.

this represents 12× to 27× overhead. An access that hits the stash takes 84 cycles (560ns).

When considering full ORAM accesses, latencies range from 19us to 30us. Much of Phantom's real-world performance is therefore determined by how much of the write-back can be overlapped with computation, but is bounded above by 23× to 50× overhead.

Compared to the ideal numbers from Section 4.6, our prototype takes an average of 4,719 cycles per full ORAM access (for a 1GB ORAM without Treetop Caching), compared to the theoretical minimum of 4,352 cycles. The difference in performance is due to the additional overhead of encryption and the latencies in the DRAM system. This relatively small overhead over the theoretical numbers shows the effectiveness of Phantom's optimizations that overlap the operation of the Path ORAM algorithm with the DRAM accesses.

7.3 DRAM Bandwidth Utilization

Figure 7.2 shows Phantom's DRAM bandwidth utilization over a series of random ORAM accesses. It shows that Phantom utilizes 93% of the theoretical peak DRAM bandwidth, i.e., the actual number of cycles between receiving the first and last words from memory relative to the number of cycles that would be taken if each memory controller returned the maximum number of bits every cycle. For reference to a practical best-case – where an application reads memory sequentially from DRAM – Phantom achieves 94% of the read bandwidth[2]. Given that Phantom accesses no additional data beyond what is required by the Path ORAM algorithm, this shows that our goal of making high use of the available bandwidth has been achieved.

7.4 FPGA Resource Usage

Figure 7.3 reports Phantom's hardware resource consumption through the percentage of logic elements (LUTs) that are used by the different configurations, as well as the number of

[2]Response data ordering was enabled for all experiments.

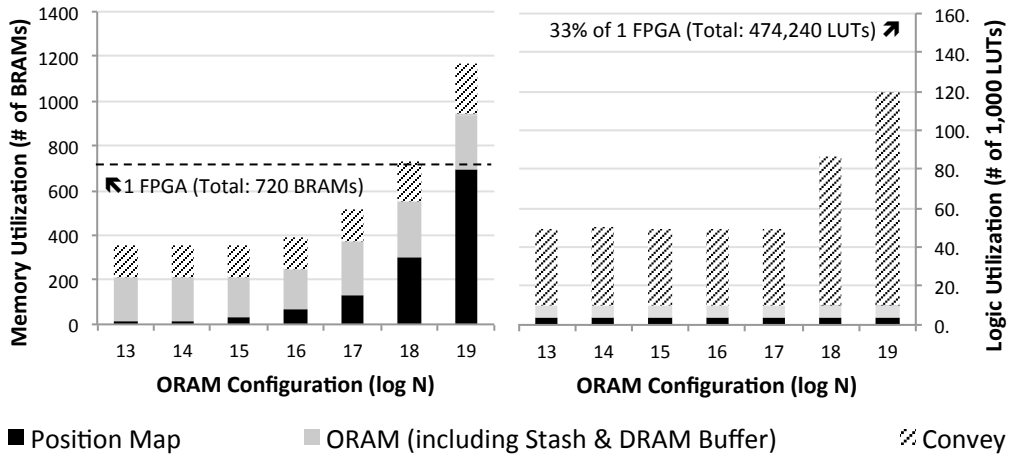


Figure 7.3: FPGA resource utilization. For sizes larger than 17 levels (1GB), the position map is stored on other FPGAs.

on-chip SRAMs (BRAMs) used on the FPGA. The design itself uses 30-52% of memory and about 2% of logic – the other resources are used by Convey's interface logic that interacts with the memory and the host CPU.

This breakdown shows (unsurprisingly) that the biggest contributors to BRAM usage are the position map and the stash. As a result, once the ORAM tree grows to more than 17 levels (1GB), the position map will not fit onto a single FPGA anymore. To support ORAM sizes larger than 1 GB, we therefore move the position map to another FPGA and communicate through (encrypted) messages over Convey's inter-FPGA communication facilities (Section 5.9).

These results do not include the resources that would be consumed by real AES crypto hardware. There exist optimized AES processor designs [39] that meet our bandwidth and frequency requirements while consuming about 22K logic elements and no BRAM – our nine required units would therefore fit onto our FPGA (as even large configurations of the ORAM controller leave almost 90% of the FPGA's logic elements available).

To give an intuition about the physical layout of Phantom, Figure 7.4 shows the floor plan of the synthesized design (including both the ORAM controller and the CPU). The figure shows how large parts of the logic are unused, but that many of the BRAMs (the vertical lines) are used up by the ORAM controller.

7.5 Impact on Application Performance

After evaluating Phantom's ORAM controller in isolation, we are now interested in how Phantom's ORAM latency translates into application slowdowns. Application performance in the presence of ORAM is extremely workload-dependent and our evaluation is far from

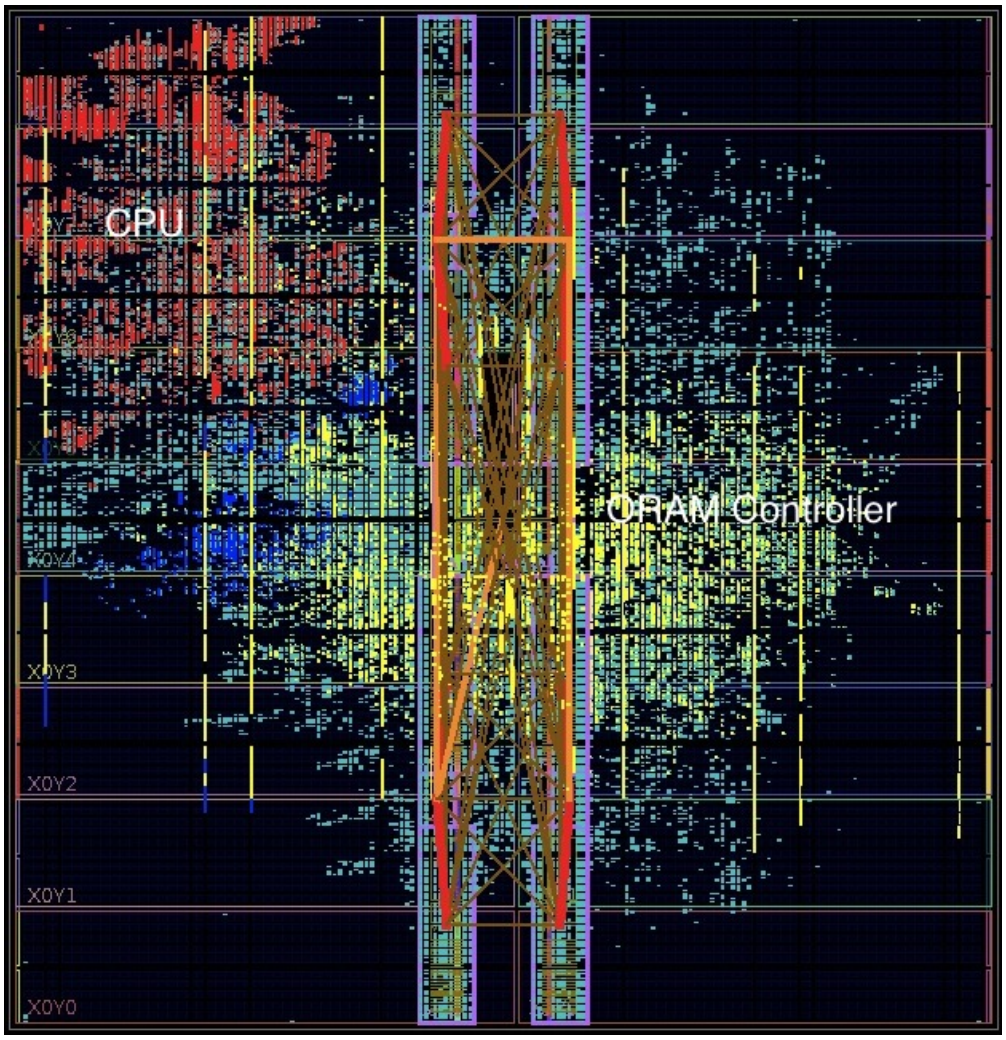


Figure 7.4: Synthesized design on a Virtex-6 LX760 FPGA.

extensive. Instead, we want to get a first-order approximation of what the performance of real applications may look like on a system like Phantom.

As an end-to-end example, we used Phantom to run the SQLite workload from our attack in Section 2.3. We ran several SQLite queries on the 7.5MB census database [20], using our real Phantom hardware (Figure 7.5). Due to the hardware prototype's extremely small cache sizes, a very large percentage of memory accesses are cache misses (we ran the same workloads on an ISA simulator with cache models and found 7.7% dcache misses, and 55.5% LLC misses). As a result, we observed slowdowns of 6.5× to 14.7× for a set of different SQLite queries (which are described in more detail later in this section). This experiment shows how crucial caching is to achieve good ORAM performance: with high miss rates, the overhead of ORAM compared to regular memory accesses leads to order-of-magnitude slow-

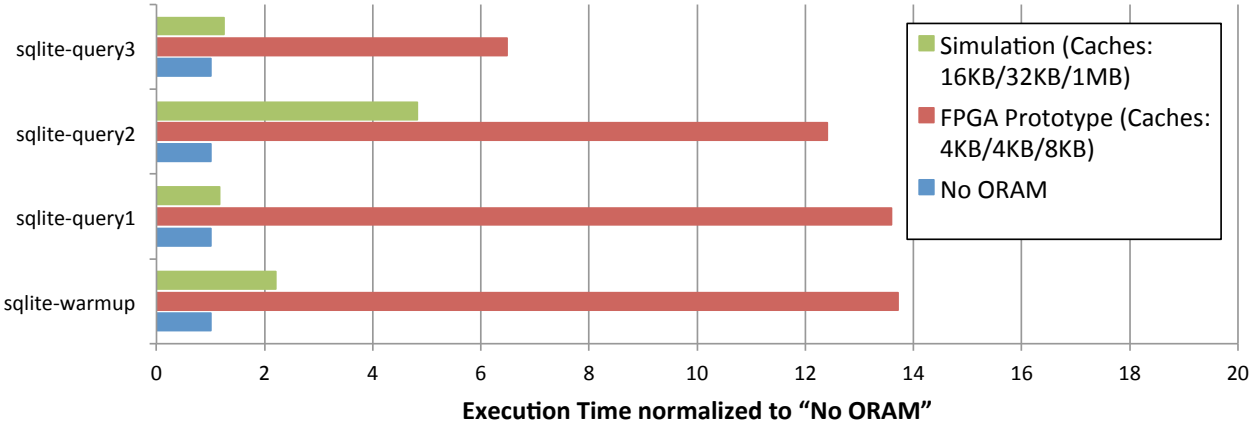


Figure 7.5: Performance numbers on our real FPGA hardware and in simulation. All numbers are for a 1GB ORAM and are normalized to the same application running on the real hardware with the RISC-V processor directly connected to the Convey memory system.

downs compared to a baseline without ORAM. While there is no fundamental reason we could not have larger caches in our hardware prototype, this would have required significant work that is largely orthogonal to the questions we set out to investigate in Phantom. We therefore chose to use simulation to investigate the impact of larger cache sizes.

We investigate the effect on applications in the presence of realistic cache sizes (namely a 16KB icache, 32KB dcache and 1MB LLC) by extending the RISC-V ISA-level simulator [87] with a timing model derived from our real Phantom prototype to simulate how our system would perform with realistic cache sizes. The model assumes an in-order pipelined RISC-V processor with 1 cycle per instruction, 2 cycles for branch misses, 2 cycles to access the data cache, 14 cycles to access the LLC, and 89 cycles to access a 128B cache line from DRAM (using our measurement for the full Convey memory system) and our access latencies for different ORAM configurations (Table 7.1).

While this approach is not cycle-accurate, it gives us an estimate of the performance we could expect on a realistic system with larger cache sizes. We chose our model parameters conservatively: we assume the processor runs at full speed (150 MHz instead of 75 MHz), and is therefore stressing the memory system more than our actual implementation would. We also assumed that the processor waits for a full ORAM request to finish before continuing execution (in reality, it can continue after the read phase) and always needs to evict a block from the ORAM buffer, causing an up to 4× longer ORAM latency than in a real execution. At the same time, we allow the baseline to use the full Convey memory system, i.e. all 16 memory-channels in parallel (albeit with only one memory request in flight – a processor tuned to Convey's very wide memory system could still achieve better performance).

We see that a real-world workload, such as SQLite, can absorb a fair amount of our memory access overheads and make use of our relatively large block size of 4KB. Figure 7.6 shows our results for applying our timing model to several SQLite queries on the 7.5MB

Model Parameters	
Processor Model	functional, no FPU, no branch prediction
ORAM Block Size	4KB (-128b block headers)
L1 Instruction Cache	16KB, 64B cache line size, 2-way set-associative
L1 Data Cache	32KB, 64B cache line size, 4-way set-associative
L2 Unified Cache	1MB, 128B cache line size, 8-way set-associative
ORAM Block Buffer	8 blocks ($8 \times 4\text{KB} = 32\text{KB}$)
Instruction Latencies	
Divide / Multiply	70 cycles
Branch taken / L1 data cache hit	2 cycles
L1 cache miss hitting in L2	14 cycles
L1 cache miss missing in L2	4 cycles + DRAM/ORAM access time
All other instructions	1 cycle
DRAM access	89 cycles
ORAM Block Buffer hit	4 cycles
13/15/17/19-level ORAM access	2839 / 3436 / 3949 / 4550 cycles

Table 7.1: Parameters of the timing model for our simulation.

census database from Figure 7.5. All four queries were executed within the same run – as such, caches survive from one query to the next. We therefore run `sqlite-query1` twice, where the first run (`sqlite-warmup`) is to warm up the caches.

Table 7.2 shows the detailed queries we are executing. `sqlite-warmup` and `sqlite-query1` are a `JOIN` operation between two large tables. The difference in execution times shows the impact of caching: in the second query, much of the tables are already in the cache and the impact of ORAM accesses is much lower. `sqlite-query2` on the other hand shows fetching a single column of a relatively small table – here, Phantom performs more poorly due to its large block size. `sqlite-query3` represents a full scan through all fields of a large table – this is an example where Phantom fares particularly well since it can make use of the locality (we would like to point out that the baseline would also benefit from pipelining requests, which we do not account for).

sqlite-query1	`SELECT zctas.zcta, zctas.population_female_total, zctas.population_male_total FROM zctas,states_zctas WHERE zctas.id=states_zctas.zcta_id AND states_zctas.state_id=29;`
sqlite-query2	`SELECT state FROM states;`
sqlite-query3	`SELECT * FROM counties;`

Table 7.2: Queries executed on the census database from [20].

In summary, our evaluation shows that the overheads added by Phantom are indeed reasonable for some real-world workloads, especially given that the fundamental cost of

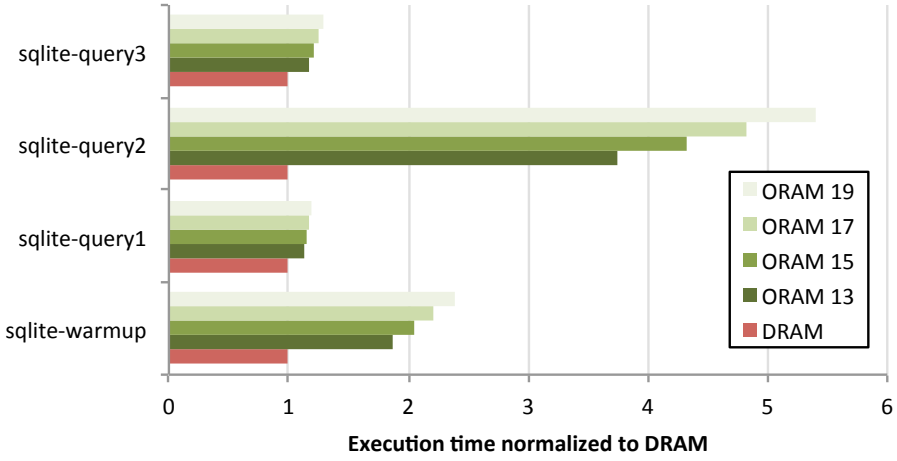


Figure 7.6: Simulated SQLite Performance. We simulated the performance of SQLite running on a timing model of PHANTOM with and without ORAM. We assume a 1MB L2 cache, 16KB icache, 32KB dcache and an extra buffer for 8 ORAM blocks (32KB).

obliviousness is a bandwidth overhead of greater than 100×. Note that these numbers depend on the application's last-level cache miss-rate and the overheads are small because most of the application's working set fits in the cache. For applications that access a very large amount of memory and exhibit little locality, the overheads can be much higher (bounded above by 50× as we saw in Section 7.2). As such, additional algorithmic and microarchitectural ORAM research is needed to bring these overheads down as well.

Chapter 8

Related Work

This chapter summarizes existing work that is related to Phantom. *In particular, it covers work on secure processors, obliviousness and Oblivious RAM. Work that is related to attacks through memory address traces is covered as well.*

8.1 Secure Processors

The concept of tamper-resistant devices that perform secure computation is as old as computing itself – for instance, early supercomputers would be protected by some form of physical security (e.g. guards to keep unauthorized users out of the room where the computer was located). Over time, the security perimeter would shrink to comprise e.g. a secure circuit board, micro-controller or authentication token (such as with smartcards).

One of the early mentions of the *secure processor* model as we consider it in the context of Phantom can be found in a 1980 paper (and associated patents) by Best [10]. This work introduces a model where not only sensitive data is encrypted but also the programs in memory. An on-chip decryption unit then transparently decrypts programs as they are being read from memory. While the original purpose of this work was to protect against piracy[1], it already introduces many of the concepts that exist in secure processors today, such as the processor as physical protection domain, and executing encrypted programs.

A unified model of secure computation was later proposed by Kuhn [48]. In this model, both code and data are encrypted in RAM and only decrypted within the security perimeter of a trusted (tamper-resistant) microprocessor chip. This model was later used by the XOM (eXecute Only Memory) project [88] to propose a microprocessor that provides compartmentalized memory for different applications and protects them from each other as well as from an external attacker with access to the machine's memory (but not the internals of the chip). Another example of a XOM-style secure processor is AEGIS [84].

A related line of work are secure co-processors such as those available from IBM [3]. While these processors provide a tamper-proof hardware platform for security-sensitive computa-

[1] Interestingly, this is the same motivation as for the original Oblivious RAM paper [30]

tion, they are less concerned about systems aspects such as multiprogramming or untrusted operating systems, and therefore differ from XOM-style secure processors.

As of today, secure processors are not widely used in mainstream computing – IBM's cryptographic co-processors find applications in mainframes and other commercial applications, and secure micro-controllers find use in security-critical embedded applications. However, there appears to be an increasing commercial interest in bringing secure processor concepts to mainstream cloud computing: Intel recently announced their Software Guard Extensions (SGX) [57], which are based on a XOM-style model and encrypt all data and code in memory while protecting applications from each other and the OS. At the same time, companies like PrivateCore [66] are attempting to provide similar guarantees in software by managing caches on commodity machines explicitly and encrypting all data in memory.

Protecting the Address Channel in Secure Processors

Since XOM and AEGIS, there has been further work on secure processors, and protecting the address channel has been one of the extensions that have been investigated. The HIDE project [100] investigated a simple approach that continuously permutes the memory space and uses a special cache management policy to prevent information leakage. While this approach is efficient (with only 1.3% overhead), it does not provide full and provable obliviousness – as such, it is difficult to ensure that no critical information leakage remains.

In contrast, Phantom aims to provide provable obliviousness, albeit at a higher performance cost. Another (concurrently) proposed secure processor design is Ascend [21]. Its high-level design has close resemblance to Phantom and proposes to use the same ORAM algorithm – Path ORAM – to provide provable obliviousness. In contrast to Phantom, Ascend only exists in high-level simulation and the authors focused on algorithmic trade-offs rather than microarchitectural details. As such, while similar on the surface, it is largely orthogonal to our work.

8.2 Oblivious RAM

Obliviousness as a formal property was first defined by Pippenger and Fischer [64] and applied to security by Goldreich in 1987 [30], who also coined the term "Oblivious RAM". The first approaches to ORAM had prohibitive overheads, but there has since been a long history of work to improve its performance. While Section 3.3 gives an overview of work in this area, here we point out some additional work that is particularly relevant to Phantom.

The Path ORAM algorithm [83] is the ORAM technique underlying Phantom. It was also investigated by the Ascend project, which looked at different algorithmic aspects, trade-offs and extensions for Path ORAM when applied to secure processors [69, 22, 70, 71].

An alternative to ORAM is work on Memory-Trace Oblivious Data Structures, which aims to achieve improved performance by exploiting data-structure-specific properties rather

than using a generalized ORAM scheme [91]. This is a more general form of domain-specific approaches to obliviousness such as in databases [2] or online advertising [9].

Another way to achieve better ORAM performance is to enable the compiler to split variables into different ORAMs, which can individually be smaller and faster to access. This requires transforming the program to prevent leakage through learning which of the ORAMs is accessed at a given time. Liu et al. present a formal type system and program transformation that achieves this goal within a custom compiler [51].

The compiler approach is particularly interesting in the context of Phantom, since it would enable Phantom to use the information available through the compiler's type system to tune itself to the given workload program (e.g., by choosing the optimal ORAM size, block size, etc). We are currently collaborating with the authors to make this possible.

8.3 Attacks through Memory Addresses

In addition to work directly related to Phantom, there is also a range of work related to our attack from Chapter 2. A large existing body of work has analyzed address traces – for varying purposes – and some of it is related to attacking programs through information leakage from their memory address trace.

As part of the HIDE project, Zhuang et al. [100] used Control Flow Graphs as fingerprints to match against a database of existing code and then used this knowledge to reconstruct the control flow of a program from captured memory address traces. This allowed them to extract cryptographic key bits from a modular exponentiation kernel, which is used in Diffie-Hellman and RSA. While very successful, this approach requires access to existing control flow graphs, which cannot be reconstructed from address traces alone. Furthermore, it requires a large portion of instruction fetches to miss in the cache, which usually requires a way to run malicious code on the secure processor.

Closely related to our statistical attack (Section 2.5), Itai et al. [41] presented initial work on classifying data structures based on address traces - for optimization purposes. They successfully used Support Vector Machines (SVMs), as well as Naïve Bayes and C4.5 classifiers and found the SVM to be the most accurate approach.

Physical Attacks

There have been a number of attacks against real systems that target the memory bus. The most prominent one is likely Huang's attack against the original Xbox [40]. He used an FPGA-based probe to tap the Xbox's HyperTransport bus and extract the Xbox's encryption keys, fully breaking the system's security. This attack shows one example how memory address traces could be extracted from a system. Alternative approaches to physical attacks (including attacks targeting busses) have been summarized by Anderson and Kuhn [1].

Side-channel Attacks in the Cloud

Protecting workloads in cloud data centers has been investigated in a number of works. Ristenpart et al. analyzed how to determine and achieve co-location with other (target) workloads in a cloud service [72], based on which Zhang et al. showed a side-channel attack [99] to classify the code paths of a different VM running on the same server, to reconstruct cryptographic keys. Their approach is based on a Hidden Markov Model and inspired ours, modeling program execution as a Markov Chain. However, they also use SVMs to classify cache behavior based on timing properties.

A similar paper by Chen et al. [13] uses a range of different side channels (such as network packet or image sizes) to reconstruct data from three real-world web applications that deal with confidential information. They model the application as a series of state transitions and keep track of an *ambiguity set* of possible states, which they reduce as they collect evidence.

Orthogonal to this work, Köpf et al. created an information theoretic model of side-channel attacks [52], which applies to the cloud as well.

Hidden Markov Models in Security

Machine Learning and Hidden Markov Models have been used quite frequently in a security context. For example, Song et al. used them to classify keystroke timing in SSH to extract passwords [79]. Intrusion detectors have used HMMs to model program behavior and detect unexpected system calls [67]. This is similar to our HMM-based attack, except that their input data is known and they only want to classify the code, not data. This work shows how the states of the Markov Chain can effectively model different program stages.

Workload Characterization and Prefetching

Some research on workload characterization has similarities to our attacks as well. For example, Sherwood et al. used k-means clustering to classify different program phases [76]. They capture the number of times each basic block was executed during a particular time interval; by randomly projecting the resulting high-dimensional vectors to a low-dimensional space and performing k-means clustering, they were able to divide the program into structurally similar phases, which allowed them to skip duplicate phases in benchmarks. This approach differs from our goals in that both data and code are known to the analyzer. On the other hand, the approach of counting basic block occurrences and using random projection on them could be essential in scaling our attack to larger programs.

A completely different application of workload characterization based on address traces is prefetching in microprocessors. Liao et al. [50] present a survey of the effectiveness of various machine learning approaches in this area.

Chapter 9

Discussion & Future Work

This chapter discusses why we believe the work on Phantom *is important and the impact that it could have in the future. It also discusses what we have learnt while building* Phantom *and summarizes promising directions for future work.*

9.1 Importance of the Attack Model

Confidentiality concerns associated with outsourcing computation are beginning to affect cloud providers. For instance, European digital policy regulation "requires all data transfers from a cloud inside the EU to a cloud maintained elsewhere to be accompanied with a notification to the data subject of such transfer and its legal effects" [11]. The US cloud industry is estimated to lose $22 billion to $35 billion in revenues (in a projected $207 billion worldwide cloud market) over the next three years due to surveillance concerns [12]. Given this fear among users, deploying a system like Phantom in data centers can enable cloud providers to fulfill the strong security requirements of cloud customers and regulators.

Physical attacks have emerged as a major threat because observing signals on a motherboard or measuring fine-grained power traces are possible without expensive equipment (examples include Huang's attack against the original Xbox [40] and attacks presented by Anderson et al. [1]). The availability of technology such as NVDIMMs shows that hardware that could enable capturing address traces from a running system can be bought off-the-shelf today. Combined with recent revelations that hardware probes are in widespread use in the intelligence community [25], the memory bus has become a very realistic attack vector.

This is confirmed by a large body of work both in the research community and industry. In principle, one could use homomorphic encryption to securely outsource computation without trusting the remote machine. However, since such schemes have many orders of magnitude slowdown over native, most industry solutions rely on trusted hardware to provide confidentiality guarantees (combined with work on tamper-evident/resistant chips and inspection resistant architectures). In the last decade, hardware manufacturers have introduced several such features, e.g., Intel's TXT and SGX instruction-set extensions to create

isolated containers, and Trusted Platform Modules (TPMs) to store sealed secrets and authenticate the hardware. While TPMs are the de-facto standard for encrypting hard-disks and attesting the software stack on a remote machine, they cannot protect against physical attacks such as a probe on the front-side bus. SGX aims to protect this channel by encrypting data on the memory bus, but does not protect the memory address trace. There are also application-specific solutions that encrypt data at rest (e.g., CryptDB [65]), but these solutions are vulnerable to address-based attacks as well. Phantom closes this vulnerability.

9.2 Why Architectural ORAM Research?

So far, ORAM research had focussed on the algorithm level and simulations. We believe it was necessary to build a synthesizable prototype of an ORAM controller to boot-strap microarchitectural research in this domain. How well ORAM algorithms translate to hardware had received little consideration before our work, while being crucial for real-world adoption. An algorithm with good theoretical overhead can fail in hardware if its operations cannot be removed from the critical path of the system and prevent it from fully utilizing the available memory bandwidth (ORAM's main bottleneck).

These aspects are not readily visible – it was necessary to design a real implementation to discover them. For example, when we first started Phantom in 2011, we began with a software simulation model using DRAMSim2, but the model missed the important detail that sorting the on-chip Path ORAM data structure on every ORAM access became a major bottleneck when scaling to a wide memory bus, causing memory stalls. As future research improves Path ORAM through optimizations, their microarchitectural impact needs to be considered to tell the real overhead in the context of secure CPUs. We believe that Phantom can raise awareness of this problem and become a platform for future ORAM research.

9.3 Using Heterogeneous Systems for Security

Phantom relies on a major trend towards domain-specific accelerators in the architecture community and shows how to apply it to an important problem in security. By targeting a platform such as the HC-2ex, where custom accelerators are deployed on FPGAs rather than ASICs, we show how such accelerators and platforms can be used for security (rather than improving performance or energy efficiency). This approach also shows a realistic deployment path for future security extensions.

One advantage of FPGA-based platforms is that they can be reconfigured to be tuned for specific workloads – we believe that this opens up many exciting research opportunities. We also think that the design patterns that Phantom uses to avoid timing leaks can help with the design of future security-related accelerators, many of which will also have to ensure not to leak timing information. Showing how to make Path ORAM work at the low frequencies that an FPGA provides is an important contribution as well.

9.4 Future Work

We see Phantom as a starting point for integrating ORAMs into real systems. While it taught us about the architectural trade-offs when implementing ORAM in hardware, and potential pitfalls that a real implementation needs to take into account, there is much more work to be done to extend it to other areas. Here, we present a subset of possible directions for follow-on work, some of which we are actively investigating at the time of writing.

Design-Space Exploration

Phantom only investigates a small part of the design space – other parameters include block size, hierarchical ORAM, multiple ORAM controllers split across multiple FPGAs on the HC-2ex (to make use of the full available memory bandwidth, which can only be achieved by using all four FPGAs) and trade-offs between different caching approaches in both the CPU and the ORAM controller. While some of these approaches have been investigated at an algorithmic level [69], there are additional trade-offs at the microarchitectural level, such as chip-area and communication cost between different FPGAs.

Our Convey prototype also provides a unique testbed for tuning the hardware ORAM implementation to specific workloads. The design space is enormous and spans both algorithmic and microarchitectural parameters. By enabling compilers to tap into this flexibility, Phantom could enable opportunities for research that may bring down ORAM overheads by another order of magnitude.

ORAM Optimizations

There are numerous optimization and extensions to Path ORAM that could be implemented in Phantom as well [69, 71]. Furthermore, Phantom does not yet prevent information leakage through the timing of ORAM accesses, which could be implemented in a future version as well. Some previous work by Fletcher et al. could be a starting point for this [22].

Compiler Support

One direction we are actively investigating are compilers and OSs that target Phantom directly as a peripheral, which can reduce application slowdowns significantly.

Liu et al. present a compiler that can transform a program to use multiple ORAM banks which can be accessed much faster than a large monolithic ORAM [51]. This technique can improve ORAM performance significantly, since it makes it possible to:

1. Only store sensitive data that *affects* the address trace in ORAM (for data that is accessed in a fixed pattern, encryption without ORAM is sufficient), and

2. Store sensitive data that is mutually non-interfering in separate, smaller ORAM banks

Their compiler directly targets PHANTOM, and we are investigating modifications to PHANTOM that enable our hardware to execute these programs efficiently without leaking additional information. Initial results are promising and show significant reduction in ORAM overheads for kernels like k-means and Dijkstra's algorithm.

Operating System Support

Since PHANTOM is designed as a co-processor, the host operating system needs to provide a way to securely share it between multiple cloud customers. While it is possible to manage PHANTOM like a traditional peripheral device, optimizations such as better context switching or batching oblivious executions may improve performance significantly.

Chapter 10

Conclusion

In this thesis, we presented PHANTOM, a secure processor with a practical oblivious memory system that achieves high performance by exploiting memory parallelism and using a microarchitecture that scales to a large number of memory channels. Importantly for adoption, we implemented PHANTOM as a prototype on the Convey HC-2ex FPGA-based platform. This makes obliviousness available today.

Our PHANTOM prototype runs on real hardware and executes real-world applications such as SQLite. During its development, we learnt about microarchitectural trade-offs related to Path ORAM that would not have been apparent in high-level simulation and can help to guide future research on ORAM. As ORAM research is starting to become of interest to the computer architecture community, we believe that such research is crucial to guide research at the algorithmic level; it is hard to evaluate something without building it.

Starting from our PHANTOM prototype, we can now investigate further design trade-offs and learn how they influence the microarchitectural level. In the future, PHANTOM will be able to expose a memory composed of DRAM, encrypted DRAM, and ORAM banks to software and thus open the door to compiler analyses that improves performance without compromising obliviousness.

Bibliography

[1] Ross Anderson and Markus Kuhn. "Tamper Resistance: A Cautionary Note". In: *Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*. WOEC'96. Berkeley, CA, USA: USENIX Association, 1996, 1–1. URL: http://dl.acm.org/citation.cfm?id=1267167.1267168 (visited on 04/25/2014).

[2] Arvind Arasu and Raghav Kaushik. "Oblivious Query Processing". In: *Proceedings 17th International Conference on Database Theory (ICDT)*. 2014, 26–37.

[3] T. W. Arnold and L.P. Van Doorn. "The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer". In: *IBM Journal of Research and Development* 48.3.4 (May 2004), pp. 475–487. ISSN: 0018-8646. DOI: 10.1147/rd.483.0475.

[4] *ArxCis-NV: Non-Volatile Memory Technology*. URL: http://www.vikingtechnology.com/arxcis-nv (visited on 04/03/2014).

[5] Amittai Aviram et al. "Determinating Timing Channels in Compute Clouds". In: *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*. CCSW '10. New York, NY, USA: ACM, 2010, 103–108. ISBN: 978-1-4503-0089-6. DOI: 10.1145/1866835.1866854. URL: http://doi.acm.org/10.1145/1866835.1866854 (visited on 04/20/2014).

[6] *AWS Case Study: Unilever*. URL: http://aws.amazon.com/solutions/case-studies/unilever/ (visited on 03/21/2013).

[7] Charles Babcock. *Cloud Connect: Netflix Finds Home In Amazon EC2*. Retrieved Sep 16, 2013. URL: http://www.informationweek.com/cloud-computing/infrastructure/cloud-connect-netflix-finds-home-in-amaz/229300547.

[8] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. New York, NY, USA: ACM, 2012, 1216–1225. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228584. URL: http://doi.acm.org/10.1145/2228360.2228584 (visited on 04/21/2014).

[9] M. Backes et al. "ObliviAd: Provably Secure and Practical Online Behavioral Advertising". In: *2012 IEEE Symposium on Security and Privacy (SP)*. May 2012, pp. 257–271. DOI: 10.1109/SP.2012.25.

[10] Robert M Best. "Preventing software piracy with crypto-microprocessors". In: *Proceedings of IEEE Spring COMPCON*. Vol. 80. 1980, 466–469.

[11] Eric Blattberg. *Europe wants to regulate the cloud for a post-Snowden world*. Oct. 2013. URL: http://venturebeat.com/2013/10/07/europe-wants-to-regulate-the-cloud-for-the-post-snowden-world/ (visited on 04/25/2014).

[12] Daniel Castro. *How Much Will PRISM Cost the U.S. Cloud Computing Industry?* Aug. 2013. URL: http://www2.itif.org/2013-cloud-computing-costs.pdf (visited on 04/03/2014).

[13] Shuo Chen et al. "Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow". In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, 191–206. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.20. URL: http://dx.doi.org/10.1109/SP.2010.20 (visited on 04/05/2014).

[14] K.-M. Chung, Z. Liu, and R. Pass. "Statistically-secure ORAM with $O(\log^2 n)$ Overhead". In: *ArXiv e-prints* (July 2013).

[15] *Convey: Better Computing for Better Analytics*. Tech. rep. Convey Computer, 2012. URL: http://www.conveycomputer.com/files/3613/5085/4052/Convey_HC-2_Product_Brochure.pdf (visited on 04/21/2014).

[16] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. "Perfectly Secure Oblivious RAM Without Random Oracles". In: *Proceedings of the 8th Conference on Theory of Cryptography*. TCC'11. Berlin, Heidelberg: Springer-Verlag, 2011, 144–163. ISBN: 978-3-642-19570-9. URL: http://dl.acm.org/citation.cfm?id=1987260.1987274 (visited on 04/04/2014).

[17] Joseph Devietti et al. "DMP: Deterministic Shared Memory Multiprocessing". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. New York, NY, USA: ACM, 2009, 85–96. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508255. URL: http://doi.acm.org/10.1145/1508244.1508255 (visited on 04/20/2014).

[18] *Engineering the next data center breakthrough: Programmable silicon platforms that bring higher value into the cloud*. Tech. rep. Xilinx, 2012. URL: http://www.xilinx.com/publications/prod_mktg/CS1078_DataCtr_ProdBrf_FINAL_HiRes.pdf (visited on 04/20/2014).

[19] Benny Evangelista. *Zynga accuses ex-manager of data theft*. Oct. 2012. URL: http://www.sfgate.com/technology/article/Zynga-accuses-ex-manager-of-data-theft-3951198.php (visited on 03/21/2014).

[20] Geoffrey Fairchild. *Update to 2010 Census SQLite Population Database*. Dec. 2011. URL: http://www.gfairchild.com/2011/12/20/update-to-2010-census-sqlite-population-database/ (visited on 04/24/2014).

[21] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. "A Secure Processor Architecture for Encrypted Computation on Untrusted Programs". In: *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*. STC '12. New York, NY, USA: ACM, 2012, 3–8. ISBN: 978-1-4503-1662-0. DOI: 10.1145/2382536.2382540. URL: http://doi.acm.org/10.1145/2382536.2382540 (visited on 04/19/2014).

[22] Christopher W. Fletcher et al. "Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-offs". In: *Proceedings of the 20th IEEE International Symposium On High Performance Computer Architecture*. HPCA '14. 2014.

[23] Mary Jo Foley. *Microsoft reveals plans for a government cloud platform*. Oct. 2013. URL: http://www.zdnet.com/microsoft-reveals-plans-for-a-government-cloud-platform-7000021654/ (visited on 03/21/2014).

[24] *FPGA Design Security Issues: Using Lattice FPGAs to Achieve High Design Security*. Tech. rep. Lattice Semiconductor, Sept. 2007. URL: http://www.latticesemi.com/~/media/Documents/WhitePapers/AG/FPGADesignSecurityIssuesUsingLatticeFPGAstoAchieveHighDesignSecurity.PDF (visited on 04/20/2014).

[25] Sean Gallagher. "Your USB cable, the spy: Inside the NSA's catalog of surveillance magic". In: *Ars Technica* (Dec. 2013). URL: http://arstechnica.com/information-technology/2013/12/inside-the-nsas-leaked-catalog-of-surveillance-magic/ (visited on 04/03/2014).

[26] *Gartner Says Cloud-Based Security Services Market to Reach $2.1 Billion in 2013*. Oct. 2013. URL: http://www.gartner.com/newsroom/id/2616115 (visited on 03/21/2014).

[27] Barton Gellman and Ashkan Soltani. "NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say". In: *The Washington Post* (Oct. 2013).

[28] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*. STOC '09. New York, NY, USA: ACM, 2009, 169–178. ISBN: 978-1-60558-506-2. DOI: 10.1145/1536414.1536440. URL: http://doi.acm.org/10.1145/1536414.1536440 (visited on 04/04/2014).

[29] Craig Gentry et al. "Optimizing ORAM and Using It Efficiently for Secure Computation". In: *Privacy Enhancing Technologies*. Ed. by Emiliano De Cristofaro and Matthew Wright. Lecture Notes in Computer Science 7981. Springer Berlin Heidelberg, Jan. 2013, pp. 1–18. ISBN: 978-3-642-39076-0, 978-3-642-39077-7. URL: http://link.springer.com/chapter/10.1007/978-3-642-39077-7_1 (visited on 04/19/2014).

[30] O. Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs". In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, NY, USA: ACM, 1987, 182–194. ISBN: 0-89791-221-7. DOI: 10.1145/28395.28416. URL: http://doi.acm.org/10.1145/28395.28416 (visited on 04/04/2014).

[31] Oded Goldreich and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *J. ACM* 43.3 (May 1996), pp. 431–473. ISSN: 0004-5411. DOI: 10.1145/233551.233553. URL: http://doi.acm.org/10.1145/233551.233553.

[32] Michael T. Goodrich and Michael Mitzenmacher. "Privacy-preserving Access of Outsourced Data via Oblivious RAM Simulation". In: *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*. ICALP'11. Berlin, Heidelberg: Springer-Verlag, 2011, 576–587. ISBN: 978-3-642-22011-1. URL: http://dl.acm.org/citation.cfm?id=2027223.2027282 (visited on 04/19/2014).

[33] Michael T. Goodrich et al. "Oblivious RAM Simulation with Efficient Worst-case Access Overhead". In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*. CCSW '11. New York, NY, USA: ACM, 2011, 95–100. ISBN: 978-1-4503-1004-8. DOI: 10.1145/2046660.2046680. URL: http://doi.acm.org/10.1145/2046660.2046680 (visited on 04/04/2014).

[34] Michael T. Goodrich et al. "Privacy-preserving Group Data Access via Stateless Oblivious RAM Simulation". In: *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '12. Kyoto, Japan: SIAM, 2012, 157–167. URL: http://dl.acm.org/citation.cfm?id=2095116.2095130 (visited on 04/04/2014).

[35] Glenn Greenwald, Ewen MacAskill, and Laura Poitras. "Edward Snowden: the whistleblower behind the NSA surveillance revelations". In: *The Guardian* (June 2013).

[36] Yufei Gu et al. "OS-Sommelier: Memory-only Operating System Fingerprinting in the Cloud". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. New York, NY, USA: ACM, 2012, 5:1–5:13. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391234. URL: http://doi.acm.org/10.1145/2391229.2391234 (visited on 04/04/2014).

[37] J. Alex Halderman et al. "Lest We Remember: Cold-boot Attacks on Encryption Keys". In: *Commun. ACM* 52.5 (May 2009), 91–98. ISSN: 0001-0782. DOI: 10.1145/1506409.1506429. URL: http://doi.acm.org/10.1145/1506409.1506429 (visited on 04/04/2014).

[38] Clemens Helfmeier et al. "Breaking and Entering Through the Silicon". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. CCS '13. New York, NY, USA: ACM, 2013, 733–744. ISBN: 978-1-4503-2477-9.

DOI: 10.1145/2508859.2516717. URL: http://doi.acm.org/10.1145/2508859.2516717 (visited on 04/04/2014).

[39] A. Hodjat and I. Verbauwhede. "A 21.54 Gbits/s fully pipelined AES processor on FPGA". In: *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004. FCCM 2004*. Apr. 2004, pp. 308–309. DOI: 10.1109/FCCM.2004.1.

[40] Andrew Huang. "Keeping Secrets in Hardware: The Microsoft Xbox(TM) Case Study". In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. 2003, pp. 355–430. URL: http://dx.doi.org/10.1007/3-540-36400-5_17 (visited on 04/25/2014).

[41] Alon Itai and Michael Slavkin. "Detecting Data Structures from Traces". In: *Workshop on Approaches and Applications of Inductive Programming*. 2007, p. 39.

[42] JASPA. *A Sparse Matrix Multiplication Benchmark*. http://www2.research.att.com/~yifanhu/SOFTWARE/JASPA/index.html.

[43] Ron Kalla et al. "Power7: IBM's Next-Generation Server Processor". In: *IEEE Micro* 30.2 (2010), pp. 7–15. ISSN: 0272-1732.

[44] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. New York, NY, USA: ACM, 2009, 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596 (visited on 04/20/2014).

[45] Boris Köpf and David Basin. "An information-theoretic model for adaptive side-channel attacks". In: *Proceedings of the 14th ACM conference on Computer and communications security*. CCS '07. Alexandria, Virginia, USA: ACM, 2007, pp. 286–296. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315282. URL: http://doi.acm.org/10.1145/1315245.1315282.

[46] *KPMG Global cloud survey: The cloud takes shape*. 2013. URL: http://www.kpmg.com/Global/en/IssuesAndInsights/ArticlesPublications/cloud-service-providers-survey/Documents/the-cloud-takes-shape-v4.pdf (visited on 03/21/2014).

[47] Tom Krazit. *Google fired engineer for privacy breach*. Sept. 2010. URL: http://news.cnet.com/8301-30684_3-20016451-265.html (visited on 03/21/2014).

[48] Markus Kuhn. *The TrustNo1 cryptoprocessor concept*. Tech. rep. 1997.

[49] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. "On the (in)Security of Hash-based Oblivious RAM and a New Balancing Scheme". In: *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '12. Kyoto, Japan: SIAM, 2012, 143–156. URL: http://dl.acm.org/citation.cfm?id=2095116.2095129 (visited on 04/04/2014).

[50] Shih-wei Liao et al. "Machine learning-based prefetch optimization for data center applications". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: ACM, 2009, 56:1–56:10. ISBN: 978-1-60558-744-8. DOI: 10.1145/1654059.1654116. URL: http://doi.acm.org/10.1145/1654059.1654116.

[51] Chang Liu, M. Hicks, and E. Shi. "Memory Trace Oblivious Program Execution". In: *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*. June 2013, pp. 51–65. DOI: 10.1109/CSF.2013.11.

[52] Rose Liu et al. "Tessellation: space-time partitioning in a manycore client OS". In: *Proceedings of the First USENIX conference on Hot topics in parallelism*. HotPar'09. Berkeley, California: USENIX Association, 2009, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1855591.1855601.

[53] Stephen Liu. *spmemvfs-0.3*. Google Code, Feb. 2010. URL: https://code.google.com/p/sphivedb/downloads/detail?name=spmemvfs-0.3.tar.gz (visited on 04/24/2014).

[54] Jacob R. Lorch et al. "Shroud: Ensuring Private Access to Large-Scale Data in the Data Center". In: *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX, 2013, 199–213. ISBN: 978-1-931971-99-7. URL: https://www.usenix.org/conference/fast13/technical-sessions/presentation/lorch.

[55] Martin Maas et al. "PHANTOM: Practical Oblivious Computation in a Secure Processor". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. CCS '13. New York, NY, USA: ACM, 2013, 311–324. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516692. URL: http://doi.acm.org/10.1145/2508859.2516692 (visited on 04/19/2014).

[56] Robert Martin, John Demme, and Simha Sethumadhavan. "TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, 118–129. ISBN: 978-1-4503-1642-2. URL: http://dl.acm.org/citation.cfm?id=2337159.2337173 (visited on 04/20/2014).

[57] Frank McKeen et al. "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. New York, NY, USA: ACM, 2013, 10:1–10:1. ISBN: 978-1-4503-2118-1. DOI: 10.1145/2487726.2488368. URL: http://doi.acm.org/10.1145/2487726.2488368 (visited on 04/04/2014).

[58] *Micron Technology, Inc., and AgigA Tech Collaborate to Develop Nonvolatile DIMM Technology*. Nov. 2012. URL: http://investors.micron.com/releasedetail.cfm?ReleaseID=721164 (visited on 04/03/2014).

[59] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746. URL: http://doi.acm.org/10.1145/1250734.1250746.

[60] Rafail Ostrovsky. "An Efficient Software Protection Scheme". In: *Proceedings on Advances in Cryptology*. CRYPTO '89. New York, NY, USA: Springer-Verlag New York, Inc., 1989, 610–611. ISBN: 0-387-97317-6. URL: http://dl.acm.org/citation.cfm?id=118209.118264 (visited on 04/19/2014).

[61] Rafail Ostrovsky and Victor Shoup. "Private Information Storage (Extended Abstract)". In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC '97. New York, NY, USA: ACM, 1997, 294–303. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258606. URL: http://doi.acm.org/10.1145/258533.258606 (visited on 04/19/2014).

[62] J Thomas Pawlowski. "Hybrid Memory Cube (HMC)". In: *Hotchips*. Vol. 23. 2011, 1–24.

[63] Benny Pinkas and Tzachy Reinman. "Oblivious RAM Revisited". In: *Proceedings of the 30th Annual Conference on Advances in Cryptology*. CRYPTO'10. Berlin, Heidelberg: Springer-Verlag, 2010, 502–519. ISBN: 3-642-14622-8, 978-3-642-14622-0. URL: http://dl.acm.org/citation.cfm?id=1881412.1881447 (visited on 04/04/2014).

[64] Nicholas Pippenger and Michael J. Fischer. "Relations Among Complexity Measures". In: *J. ACM* 26.2 (Apr. 1979), 361–381. ISSN: 0004-5411. DOI: 10.1145/322123.322138. URL: http://doi.acm.org/10.1145/322123.322138 (visited on 04/19/2014).

[65] Raluca Ada Popa et al. "CryptDB: Protecting Confidentiality with Encrypted Query Processing". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. New York, NY, USA: ACM, 2011, 85–100. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043566. URL: http://doi.acm.org/10.1145/2043556.2043566 (visited on 05/16/2014).

[66] *PrivateCore*. URL: http://www.privatecore.com/ (visited on 04/03/2014).

[67] Y. Qiao et al. "Anomaly intrusion detection method based on HMM". In: *Electronics Letters* 38.13 (June 2002), pp. 663–664. ISSN: 0013-5194. DOI: 10.1049/el:20020467.

[68] A.E. Raftery. "A model for high-order Markov chains". In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1985), pp. 528–539.

[69] Ling Ren et al. "Design space exploration and optimization of path oblivious RAM in secure processors". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. New York, NY, USA: ACM, 2013, 571–582. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485971. URL: http://doi.acm.org/10.1145/2485922.2485971 (visited on 10/15/2013).

[70] Ling Ren et al. "Integrity verification for path Oblivious-RAM". In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2013, pp. 1–6. DOI: 10.1109/HPEC.2013.6670339.

[71] Ling Ren et al. *Unified Oblivious-RAM: Improving Recursive ORAM with Locality and Pseudorandomness*. Tech. rep. 205. 2014. URL: http://eprint.iacr.org/2014/205 (visited on 04/21/2014).

[72] Thomas Ristenpart et al. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. Chicago, Illinois, USA: ACM, 2009, pp. 199–212. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653687. URL: http://doi.acm.org/10.1145/1653662.1653687.

[73] Brian Rogers et al. "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly". In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, 183–196. ISBN: 0-7695-3047-8. DOI: 10.1109/MICRO.2007.44. URL: http://dx.doi.org/10.1109/MICRO.2007.44 (visited on 04/04/2014).

[74] *SecureState Case Study: Physical Penetration, Hosting*. URL: http://www.securestate.com/Downloads/Profiling/Case-Study-Physical-Pen-Hosting-Provider.pdf (visited on 04/03/2014).

[75] Yi Shan et al. "FPMR: MapReduce Framework on FPGA". In: *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '10. New York, NY, USA: ACM, 2010, 93–102. ISBN: 978-1-60558-911-4. DOI: 10.1145/1723112.1723129. URL: http://doi.acm.org/10.1145/1723112.1723129 (visited on 04/21/2014).

[76] Timothy Sherwood et al. "Automatically characterizing large scale program behavior". In: *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ASPLOS-X. San Jose, California: ACM, 2002, pp. 45–57. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605403. URL: http://doi.acm.org/10.1145/605397.605403.

[77] Elaine Shi et al. "Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost". In: *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*. 2011.

[78] Sean W. Smith. "Outbound Authentication for Programmable Secure Coprocessors". In: *Proceedings of the 7th European Symposium on Research in Computer Security*. ESORICS '02. London, UK, UK: Springer-Verlag, 2002, 72–89. ISBN: 3-540-44345-2. URL: http://dl.acm.org/citation.cfm?id=646649.699490 (visited on 04/04/2014).

[79] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. "Timing analysis of keystrokes and timing attacks on SSH". In: *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*. SSYM'01. Washington, D.C.: USENIX Association, 2001, pp. 25–25. URL: http://dl.acm.org/citation.cfm?id=1267612.1267637.

[80] Emil Stefanov and Elaine Shi. "ObliviStore: High Performance Oblivious Cloud Storage". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, 253–267. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.25. URL: http://dx.doi.org/10.1109/SP.2013.25 (visited on 04/05/2014).

[81] Emil Stefanov and Elaine Shi. "Path O-RAM: An Extremely Simple Oblivious RAM Protocol". In: *CoRR* abs/1202.5150 (2012).

[82] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. "Towards Practical Oblivious RAM". In: *NDSS*. 2012.

[83] Emil Stefanov et al. "Path ORAM: An Extremely Simple Oblivious RAM Protocol". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. CCS '13. New York, NY, USA: ACM, 2013, 299–310. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516660. URL: http://doi.acm.org/10.1145/2508859.2516660 (visited on 04/04/2014).

[84] G. Edward Suh et al. "AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing". In: *Proceedings of the 17th Annual International Conference on Supercomputing*. ICS '03. New York, NY, USA: ACM, 2003, 160–171. ISBN: 1-58113-733-8. DOI: 10.1145/782814.782838. URL: http://doi.acm.org/10.1145/782814.782838 (visited on 04/04/2014).

[85] G. Edward Suh et al. "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions". In: *Proceedings of the 32nd annual international symposium on Computer Architecture*. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 25–36. ISBN: 0-7695-2270-X. DOI: 10.1109/ISCA.2005.22. URL: http://dx.doi.org/10.1109/ISCA.2005.22.

[86] Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780136006633.

[87] *The RISC-V Instruction Set Architecture*. URL: http://riscv.org/ (visited on 04/04/2014).

[88] David Lie Chandramohan Thekkath et al. "Architectural Support for Copy and Tamper Resistant Software". In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IX. New York, NY, USA: ACM, 2000, 168–177. ISBN: 1-58113-317-0. DOI: 10.1145/378993.379237. URL: http://doi.acm.org/10.1145/378993.379237 (visited on 04/04/2014).

[89] Mohit Tiwari et al. "Complete Information Flow Tracking from the Gates Up". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. New York, NY, USA: ACM, 2009, 109–120. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508258. URL: http://doi.acm.org/10.1145/1508244.1508258 (visited on 04/20/2014).

[90] Adam Waksman and Simha Sethumadhavan. "Silencing Hardware Backdoors". In: *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, 49–63. ISBN: 978-0-7695-4402-1. DOI: 10.1109/SP.2011.27. URL: http://dx.doi.org/10.1109/SP.2011.27 (visited on 04/20/2014).

[91] Xiao Wang et al. *Oblivious Data Structures*. Tech. rep. 185. 2014. URL: http://eprint.iacr.org/2014/185 (visited on 04/25/2014).

[92] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Tech. rep. UCB/EECS-2011-62. EECS Department, University of California, Berkeley, May 2011. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html.

[93] Peter Williams, Radu Sion, and Bogdan Carbunar. "Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. New York, NY, USA: ACM, 2008, 139–148. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455790. URL: http://doi.acm.org/10.1145/1455770.1455790 (visited on 04/19/2014).

[94] Peter Williams, Radu Sion, and Alin Tomescu. "PrivateFS: A Parallel Oblivious File System". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. New York, NY, USA: ACM, 2012, 977–988. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382299. URL: http://doi.acm.org/10.1145/2382196.2382299 (visited on 04/05/2014).

[95] Rafal Wojtczuk and Alexander Tereshkin. "Attacking Intel BIOS". In: *BlackHat*. Las Vegas, USA, 2009.

[96] Sophie Woodman. *Gartner Predict Cloud Computing Spending to Increase by 100% in 2016, Says AppsCare*. July 2012. URL: http://www.prweb.com/releases/2012/7/prweb9711167.htm (visited on 03/21/2014).

[97] Meg Yarcia. *NGOs and the Cloud*. Apr. 2013. URL: http://www.techsoup.org/support/articles-and-how-tos/ngos-and-the-cloud (visited on 03/21/2014).

[98] G.L. Zhang et al. "Reconfigurable acceleration for Monte Carlo based financial simulation". In: *2005 IEEE International Conference on Field-Programmable Technology, 2005. Proceedings*. Dec. 2005, pp. 215–222. DOI: 10.1109/FPT.2005.1568549.

[99] Yinqian Zhang et al. "Cross-VM side channels and their use to extract private keys". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 305–316. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382230. URL: http://doi.acm.org/10.1145/2382196.2382230.

[100] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. "HIDE: an infrastructure for efficiently protecting information leakage on the address bus". In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS-XI. Boston, MA, USA: ACM, 2004, pp. 72–84. ISBN: 1-58113-804-0. DOI: 10.1145/1024393.1024403. URL: http://doi.acm.org/10.1145/1024393.1024403.

Image Sources

The PHANTOM logo on the front page is public domain and was taken from:
http://www.clipartpal.com/clipart_pd/holiday/halloween/phantom_10492.html

Use of Previously Published or Co-Authored Material

This thesis is directly based on the following paper:

PHANTOM: Practical Oblivious Computation in a Secure Processor. Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanović, John Kubiatowicz, Dawn Song. ACM Conference on Computer and Communications Security (CCS '13), Berlin, Germany, November 2013

The majority of material in this thesis is directly taken from this paper which includes, but is not limited to, text, figures and experimental results. Most of the text has been edited to include additional details and improve the wording or flow. Furthermore, in accordance with university regulations, the following steps have been taken:

- Permission for using any of the material from the above paper within this thesis has been requested and was granted by the Dean of the Graduate Division at UC Berkeley (using the form "Obtaining Permission to Include Previously Published or Co-authored Material", approval received on October 10, 2013).
- Permission to use any material from the above paper by all co-authors of the paper has been received in writing (e-mail).
- The material of the paper has been "incorporated into a larger argument that binds together the whole thesis"[1]

While the final editing pass is my own work, the material of the original paper was a close collaboration between all the authors and text in particular was written as a collaborative effort: all authors had access to the Github repository for the paper, and it is impossible to distinguish which author wrote which block of text or created which figure (in particular since most parts of the paper underwent multiple editing passes by different authors, or were written collaboratively while in the same room).

[1] `https://grad.berkeley.edu/policies/guides/thesis-filing/`

As such, I want to explicitly clarify that I do not claim sole credit for the work in this thesis (including both the writing and the implementation of PHANTOM). However, my goal was for this thesis to describe the work for which I have contributed the *majority* of the work. Furthermore, I tried to explicitly point out contributions of the other authors, even though this was not always possible for practical reasons.

Material from the CCS'13 paper has also been used in adapted form in the following two submissions (none of which count as formally published, since the first is an informal workshop and the other a rejected journal submission). Some of the adapted text has been used in this thesis as well (but is assumed to be covered by the initial permission for using co-authored material from the original paper).

- **A High-Performance Oblivious RAM Controller on the Convey HC-2ex Heterogeneous Computing Platform**. Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanović, John Kubiatowicz, Dawn Song. Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL '13), Davis, CA, December 2013
- **PHANTOM: Practical Oblivious Computation in a Secure Processor**. Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanović, John Kubiatowicz, Dawn Song. Submission to Micro Top Picks 2013 (Rejected)

This thesis also contains material from class reports for CS250, Fall 2011 (co-authored with Eric Love, basis for the CCS'13 paper) and CS281a, Fall 2012 (no co-authors).

Funding Information

The following funding information covers all authors of the original Phantom *paper at CCS'13 and is not restricted to the author of this thesis.*

This work partially supported by Microsoft (Award #024263), Intel (Award #024894) and matching U.C. Discovery funding (Award #DIG07-10227); by National Science Foundation (NSF) Grant #1136996 to the Computing Research Association for the CIFellows Project; by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; by the NSF Graduate Research Fellowship under Grant No. DGE-0946797; the DoD National Defense Science and Engineering Graduate Fellowship; a Google Research Award and a grant from the Amazon Web Services in Education program; the NSF under Grants No. CNS-1314857, CCF-0424422, 0311808, 0832943, 0448452, 0842694, 0627511, 0842695, 0808617, 0831501 CT-L, the Air Force Office of Scientific Research under MURI Award No. FA9550-09-1-0539, the Air Force Research Laboratory under grant No. P010071555, the Office of Naval Research under MURI Grant No. N000140911081, and by the MURI program under AFOSR Grant No. FA9550-08-1-0352. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.