PHANTOM: Practical Oblivious Computation in a Secure Processor

Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari[‡],
Elaine Shi[†], Krste Asanović, John Kubiatowicz, Dawn Song

University of California, Berkeley [‡] University of Texas, Austin [†] University of Maryland, College Park

ABSTRACT

We introduce PHANTOM[1], 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 or ORAM. We first improve an existing ORAM algorithm and construct an empirical model for its trusted storage requirement. We then present PHANTOM, an oblivious processor whose novel memory controller aggressively exploits DRAM bank parallelism to reduce ORAM access latency and scales well to a large number of memory channels. Finally, we build a complete hardware implementation of PHANTOM on a commercially available FPGA-based server, and through detailed experiments show that PHANTOM is efficient in both area and performance. Accessing 4KB of data from a 1GB ORAM takes 26.2us (13.5us for the data to be available), a 32× slowdown over accessing 4KB from regular memory, while SQLite queries on a population database see $1.2-6\times$ slowdown. PHANTOM is the first demonstration of a practical, oblivious processor and can provide strong confidentiality guarantees when offloading computation to the cloud.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Miscellaneous

Keywords

Secure Processors; Oblivious RAM; FPGAs; Path ORAM

1. INTRODUCTION

Confidentiality of data is a major concern for enterprises and individuals who wish to offload computation to the cloud. In particular, cloud operators have physical access to machines and can observe sensitive information (data and code) as it moves between a CPU and physical memory [17, 19, 42]. In response to such attacks, commercial interest in protecting off-chip data has begun to grow [1].

[1]PHANTOM stands for **P**arallel **H**ardware to make **A**pplications **N**on-leaky **T**hrough **O**blivious **M**emory.

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516692.

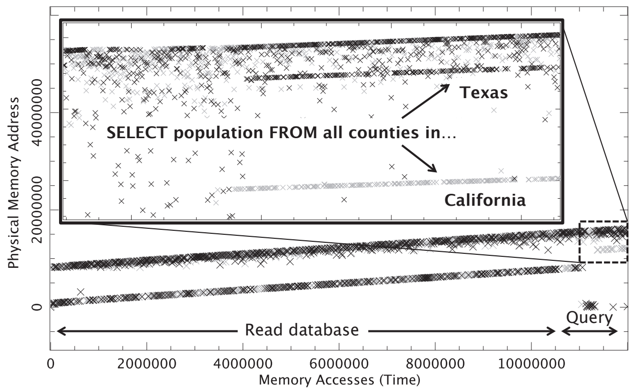

Figure 1: Visible information leakage in memory address traces from SQLite, running on a RISC-V processor model with caches. Two queries running on the same SQLite database yield clearly discernible memory accesses.

To protect against such attacks, prior work has proposed secure processors [27, 29, 33, 34] that automatically encrypt and integrity-check all data outside the processor – whether in DRAM or non-volatile storage.

Although secure processors encrypt memory contents, off-the-shelf DRAMs require that *memory addresses* be transmitted over the memory bus in cleartext. An attacker with physical access can snoop the memory bus and observe the locations of RAM accessed [19] and in turn learn sensitive data such as encryption keys or information about user-level programs [42] and guest VMs in a virtualized server [15].

As an example, we demonstrate how a program is susceptible to information leaks through the address bus. Figure 1 plots memory accesses over time for a simulated processor running two different SQLite queries – to select Californian citizens v. Texas citizens – on a population census database. The two queries generate a *visually* distinct address trace, even when the effects of on-chip caches and SQLite instruction fetches are accounted for.

Preventing such information leakage requires making memory address traces computationally indistinguishable, or *oblivious.* In this paper, we propose a new hardware architecture for efficient oblivious computation that ensures both *data confidentiality* and *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 DRAM locations accessed.

Oblivious RAM: We rely on an algorithmic construct called Oblivious RAM (ORAM), initially proposed by Goldreich and Ostrovsky [11], and later improved in numerous subsequent works [12, 14, 38]. Intuitively, ORAM techniques obfuscate memory access patterns through random permutation, reshuffling, and reencryption of memory contents, and 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.* [32] – a simple algorithm with a high degree of memory access parallelism. Path ORAM builds on the new binary-tree ORAM framework recently proposed by Shi *et al.* [28].

Other recent work has also used Path ORAM to propose a secure processor (ASCEND [9, 26]); this 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 building a practical oblivious system – complete with a CPU, an ORAM controller, and running non-trivial programs like SQLite running 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 building a practical oblivious system.

Challenges: Making oblivious processors practical poses several challenges. The first is Path ORAM's significant memory bandwidth overhead – more than 100× over a non-secure access. Second, Path ORAM's irregular, data-driven nature makes it difficult to simply add more memory channels and build a deterministic yet efficient ORAM controller. Finally, we do not propose a custom chip but rely on an off-the-shelf FPGA platform that, on the one hand, can provide high memory bandwidth but on the other hand 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 more memory channels even harder.

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

1. Empirical Model for Path ORAM. We determine Path ORAM's trusted memory requirements by simulating both synthetic, worst-case memory access traces and SPEC benchmark traces. We develop an empirical model for Path ORAM's trusted memory size v. size of the ORAM, and show that SPEC benchmarks' high degree of memory locality means that they require considerably less trusted memory than the worst-case access traces.

2. Efficient ORAM Microarchitecture. We introduce an ORAM controller architecture that is very effective at utilizing high-bandwidth DRAM – 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 1 memory controller.

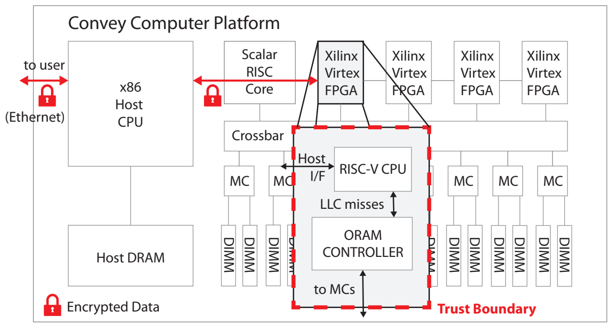

Figure 2: Phantom prototype on a Convey HC-2ex Computing Platform. PHANTOM comprises a CPU, non-volatile memory, and an ORAM controller implemented on a single chip (here, an FPGA). All *digital* signals outside the PHANTOM chip are assumed to be visible to the adversary.

3. Real-world evaluation. We build and evaluate PHANTOM's oblivious memory controller on an FPGA-based computing platform running SQLite workloads, and simulate its performance for different cache sizes. Using several ORAM configurations, we show that PHANTOM logic requires only 2% of the logic 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 to a 1GB ORAM – which translates to 0.2× to 5× slowdown for SQLite queries.

To the best of our knowledge, this is the first practical demonstration of an oblivious processor.

2. PLATFORM OVERVIEW

2.1 Usage Model

We consider a setting where a *user* wants to offload sensitive data and computation to the cloud, a *cloud provider* sells infrastructure as a service, and a *hardware manufacturer* creates secure FPGAs.

We envision a secure processor that has non-volatile memory on-chip to store a unique private key. 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 processors 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 is complete.

PHANTOM is our take at this secure processor. It supports two usage scenarios: 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. 2) 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 [1]. When sensitive data must be spilled off-chip, the trusted program makes encrypted ORAM reads/writes through the PHANTOM coprocessor. The second scenario is harder to make verifiably secure because preventing plain-text data from going off-chip on a commercial microprocessor is complicated, but has the benefit that existing applications can be ported easily and can run faster than on the FPGA.

2.2 Attack Model

We aim to protect against untrusted cloud providers. We trust users and hardware manufacturers (malicious hardware attacks [36] are out of scope) and focus on digital attacks where the cloud provider probes board-level interconnect and chip pins to learn confidential data. While existing secure processors prevent explicit data leakage with encryption, we prevent *implicit* information leakage through the address bus. Specifically, we provide the property that "for any two data request sequences of the same length, their access patterns are computationally indistinguishable by anyone but the client" (from definition of ORAM security [31]).

Note that the *total* execution time (i.e. a termination channel) is out of scope for ORAM. 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 classified data. 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. 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).

Our current implementation does not ensure *integrity* of data, but the Path ORAM tree can be treated as a Merkle tree to efficiently provide integrity with freshness [25, 32]. 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 [20,35], obfuscation [23], and deterministic execution [4, 8].

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 access by the secure CPU. Alternate obfuscation approaches [23] exist as well.

2.3 FPGAs: Opportunities & Challenges

We implemented our ORAM system on a Convey HC-2ex server (Figure 2). 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) and 16 independent memory channels, each with 64 DRAM banks, for a combined memory of 64GB. Convey's combination of reconfigurable logic and high bandwidth memory system presents an opportunity to make the memory bandwidth-hungry ORAM algorithm practical.

Designing high performance logic on an FPGA, however, is challenging. FPGAs operate at much lower frequencies than custom chips – for instance, a simple RISC CPU runs at only 75MHz, and the PHANTOM controller runs at 150MHz – because logic is implemented as a network of interconnected look-up tables. Our task thus is 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. Sections 4 and 5 describe how we achieve these requirements.

3. THE PATH ORAM ALGORITHM

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 encrypting the accessed content with a different nonce at every access). Furthermore, we assume the secure processor has a small amount of trusted memory, which the ORAM controller can access without revealing any information to the attacker. This memory is used to keep track of where data resides in untrusted 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, the controller 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 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), the ORAM controller writes the same path back to memory. Since a requested block is reassigned to a random leaf node before it is written back to untrusted memory, it may belong to a different path from that on which it was read. Since 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 it is written back – this is called *reordering*. At the same time, 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 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 illustrates its execution.

3.1 Algorithmic Details

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*.

All data stored in untrusted memory has to be encrypted at all times, and is reencrypted with a different nonce 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

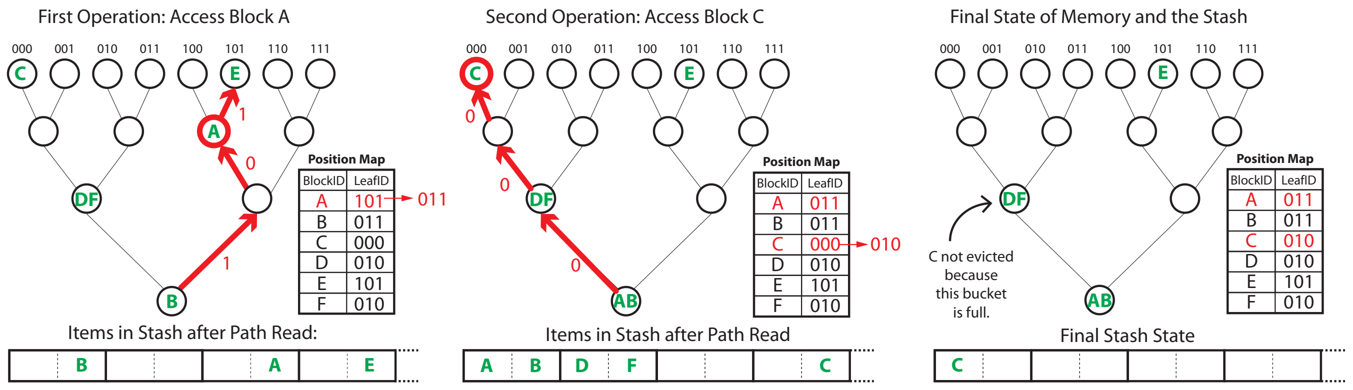
First Operation: Access Block A
Second Operation: Access Block C
Final State of Memory and the Stash
000
001
010
011
100
101
110
111
Position Map
BlockID
LeafID
A 101 → 011
B 011
C 000
D 010
E 101
F 010
A 011
B 011
C 000 → 010
D 010
E 101
F 010
A 011
B 011
C 010
D 010
E 101
F 010
C not evicted because this bucket is full.
Items in Stash after Path Read:
Items in Stash after Path Read
Final Stash State

Figure 3: The Path ORAM Algorithm from [32]. 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.

time the block is accessed, and affects the encryption of the entire block.

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.

Even with reordering, there can be cases where not all blocks in the stash can be written back to the current path (Figure 3). 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 Section 3.2 we present an empirical analysis to determine a stash size that makes these overflows extremely unlikely.

Access timing: 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. As described in Section 2.2, this is orthogonal to our microarchitecture (and hence not implemented) but would be required in a real deployment.

Algorithm 1 Pseudo-code of Path ORAM [32]

```
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)
```

Fundamental Overheads: Path ORAM's obliviousness has both space and bandwidth costs.

The capacity N of the ORAM is defined as the number of logical blocks it can store. We set the number of leaf nodes in the tree to $N/4$, and the number of blocks in each bucket to 4. As a result, 50% of the physical memory is available as oblivious memory (including data and 0.4% for block headers) and the rest is reserved for dummy blocks.

Since each block access results in an entire path read and write, Path ORAM's bandwidth overheads range from $104\times$ for a 13-level ORAM tree (16MB capacity with 4KB blocks) to $192\times$ for a 24-level tree (32GB 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 read two orders of magnitude more data per memory access.

3.2 Empirical Stash Size Analysis

To implement Path ORAM, we first had to determine a suitable stash size that makes stash overflows unlikely.

Our goal was to find a stash size such that the overflow probability is $2^{-\lambda}$ for a suitable value of λ, e.g. $\lambda = 128$. One way to do this is to rely on a recent theoretic bound [32]

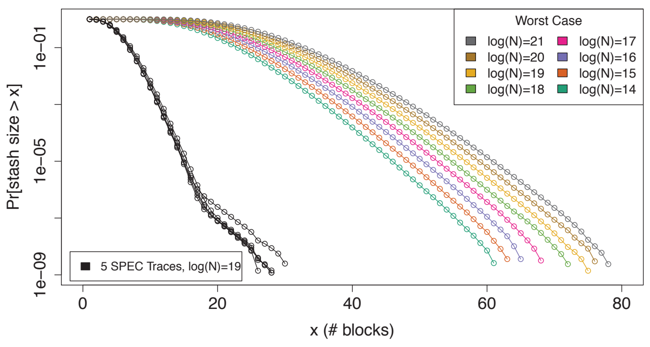

Figure 4: Distribution of stash sizes for SPEC and worst-case traces. N is the size of the ORAM in blocks. Unlike in the Path ORAM paper [32], the stash size measured includes the temporarily fetched path during Access.

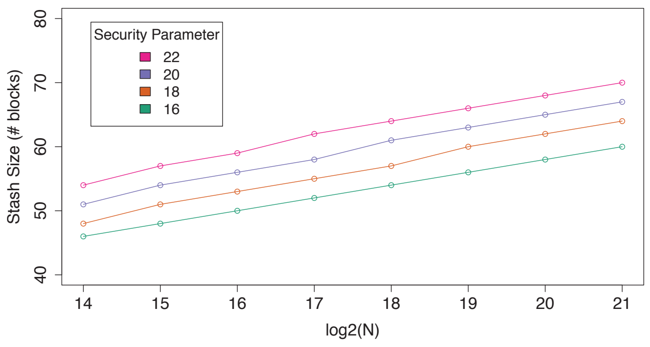

Figure 5: Empirical measurements of minimum stash size such that the probability of the stash overflowing is bounded by $2^{-\lambda}$ where λ is the security parameter. Unlike in the Path ORAM paper [32], the stash size measured includes the temporarily fetched path during Access.

on Path ORAM. However, this theoretic bound is only tight up to a constant factor.

We performed an empirical analysis that determines the required size of the stash with high confidence. Our analysis is based on a number of long simulation runs of the ORAM algorithm (reassigning ORAM blocks to random leaf nodes at every access)[2].

We run the algorithm for billions of time steps, and measure the fraction of time the stash has size at most x, for varying values of x. The purpose is to determine the stash size for a given overflow probability of $2^{-\lambda}$. We should set λ to be a large enough (say, 80). However, we cannot simulate far enough to $\lambda = 80$, since such negligible overflow probabilities are computationally infeasible to directly measure. We therefore simulate for smaller λ values and extrapolate the stash size for large λs based on the theoretic analysis of Path ORAM [32].

We used two different kind of memory access traces as input for this simulation. 1) *Worst-case traces* that exhibit no locality (i.e. blocks $1, 2, \ldots$ are accessed sequentially, wrapping around after the last block): this represents the worst-case input for the stash size [32] since accessing a block that is not already in the stash will cause more blocks to be read into the stash; and 2) *SPEC traces*: real-world memory access traces that exhibit high locality. We therefore expect to see lower stash consumption compared to worst-case traces. The benchmarks we ran were *mcf*, *cactus*, *gemsfdtd*, *lesli3d* and *soplex* with an ORAM block size of 4KB. We chose these benchmarks as they had a large number of cache misses and would thus stress the stash. While we choose our implementation's stash size conservatively based on the worst-case traces, the SPEC traces also shed light on how differently real-world traces behave in comparison with the worst-case.

We start with an initially empty ORAM and first perform N accesses to place all N blocks within the Path ORAM tree. Then we perform another 100 million round-robin (worst-case) memory accesses to warm up the ORAM. For the worst-case traces, we then simulate 5 billion memory accesses and record the stash size each time the stash content changes. The SPEC traces have 1.6 billion to 11 billion memory accesses.

[2]It is well-known that for regenerative stochastic processes, time average is equal to ensemble average — therefore, a single long run is as good as many shorter runs.

For each trace, Figure 4 plots the fraction of accesses during which the stash size exceeded a particular value. Each of the values on the y-axis corresponds to a particular probability $2^{-\lambda}$ that the stash will overflow at any given access, where we call λ the *security parameter*. In practice, a good value for λ may be 128. For large enough values of λ, we should not expect to see any stash overflows in any reasonable amount of time (just like encryption cannot be broken in any reasonable amount of time).

Using the data from Figure 4, we derived the minimum stash size that makes the probability of overflowing $2^{-\lambda}$ for a fixed λ, varying the size of the ORAM (N). The results are shown in Figure 5. The figure suggests that for a given security parameter, the stash size grows linearly in $\log N$.

Based on the findings in Figures 4 and 5, and the closed form theoretic bound in [32], using least square fitting, we determined the following minimum stash size (here the stash counts both the temporarily fetched path during access and the overflowing blocks):

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

with an R^2 value of 0.9973.

From the known theoretic bound on Path ORAM [32], we know that the above stash size includes two parts: 1) a $O(\log N)$ part for storing the path fetched from the server; and 2) a $O(\lambda)$ part for storing overflowing blocks after the write-back phase of each ORAM access. 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 security parameter values from $\lambda = 65$ to $\lambda = 143$.

4. THE ORAM CONTROLLER

In this section, we discuss how PHANTOM achieves high performance compared to a naïve Path ORAM implementation without breaking its security guarantees.

Recalling Section 2, PHANTOM receives an encrypted program and data as input, and produces encrypted data as output. PHANTOM's CPU runs the program and generates an address to access from the ORAM on each last-level cache miss. The ORAM controller then translates this address into a leaf node, reads and decrypts the path to the leaf, and returns a value to the CPU or updates the appropriate block. It then encrypts and writes back the entire path.

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 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 adds novel sorting stage to Path ORAM; we expect it to be 8× better than the baseline.

4.1 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). Path ORAM's bandwidth requirement thus motivates the use of platforms such as Convey HC-2ex that have wide memory systems: 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 naïve implementation. However, exploiting the higher memory bandwidth is non-trivial: it is necessary to co-optimize the hardware implementation and the Path ORAM algorithm itself.

We now present the key design decisions of our implementation. Table 1 summarizes the potential performance gains over a naïve implementation. We call these numbers 'potential' because the analysis assumes a perfect PHANTOM implementation without DRAM stalls. We demonstrate experimentally in Section 6 that PHANTOM comes close to this ideal on an actual FPGA implementation.

Table 1 presents three different ORAM configurations and a baseline design that uses a standard, 128 bit 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 block. 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 present a precise 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 a 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).

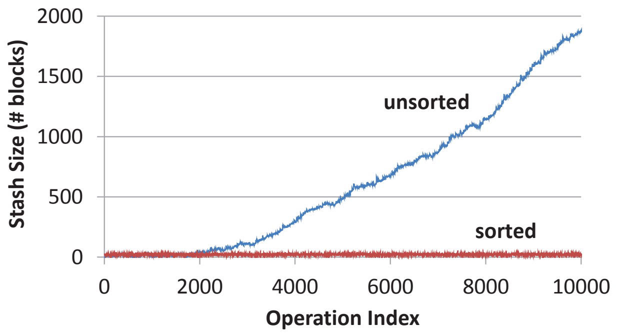

Figure 6: Benefits of sorting the stash contents before each write phase ($N=2^{19}$). Our heap-based reordering data structure ensures that we can sort the stash efficiently without stalling the ORAM. If we instead tried to use an efficient (O(1)) stash eviction strategy that doesn't perform sorting, the stash size will grow uncontrollably. Note: unlike in the Path ORAM paper [32], the stash size measured here includes the temporarily fetched path.

The latter is of particular importance: The ORAM controller has to find the block that can be placed deepest into the current path, and do so while the memory controllers push in 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 (starting from the bottom), in parallel with writing to memory. 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 2× the potential performance improvement with stash size $C = 128$ (Table 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. To verify whether a simpler alternative would suffice, we analyze an $O(1)$ option: choosing from the top 4 entries in a list of ORAM blocks to be written back, instead of considering the entire stash to find the block that can be pushed the furthest down the current path. Our analysis – in Figure 6 – shows that this simple scheme to choose blocks for write-back causes the stash to grow uncontrollably.

We hence propose a different approach – to split the task of picking the next block to write to the current path into two phases: a *sorting phase* that sorts blocks by how far they can be moved down the current path, and a *selection stage* that (during writeback) looks at the last block in the sorted list and checks in one cycle whether it can be written back into the current position or not – if not, no other block can, and we have to write a dummy.

We further improve on this approach by replacing the sorting and selection stages by a min-heap This replaces an $O(C \log C)$ operation by two $O(\log C)$ operations, where C is the size of the stash. This makes it now possible to overlap sorting completely with the path read and selecting with the path write phase.

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 k 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 12).

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 on-chip memory module on an FPGA (i.e a Block RAM or BRAM) is limited to 2 read/write ports. However, BRAMs in PHANTOM that constitute the stash have to be multiplexed among four functional units (encryption, decryption and reads/writes by the secure CPU). We 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 at any particular clock cycle. Implementing the heap to reorder stash entries also requires similar tricks which we describe in Section 5.6.

FPGAs also impose strict timing constraints on our design. Convey's DRAM controllers operate at 150MHz which makes this frequency the minimum target for PHANTOM. We modified a baseline PHANTOM implementation extensively to replicate and pipeline critical paths, and implemented PHANTOM's RISC CPU on a separate clock domain (75MHz) from the ORAM controller to meet timing constraints.

4.2 Preserving Security

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 time. 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 7 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

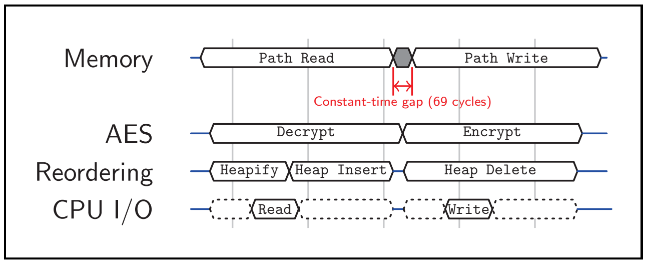

Figure 7: Overlapping the different steps of the Path ORAM algorithm with the memory accesses.

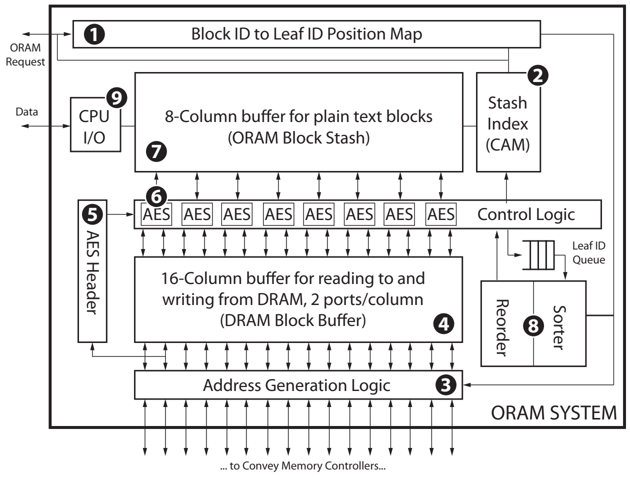
ORAM Request
Data
❶ Block ID to Leaf ID Position Map
❷ Stash Index (CAM)
❾ CPU I/O
8-Column buffer for plain text blocks (ORAM Block Stash)
❼
❻ AES AES AES AES AES AES AES AES Control Logic
❺ AES Header
Leaf ID Queue
16-Column buffer for reading to and writing from DRAM, 2 ports/column (DRAM Block Buffer)
❹
Reorder
❽
Sorter
Address Generation Logic
❸
ORAM SYSTEM
... to Convey Memory Controllers...

Figure 8: Overview of the ORAM system.

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.

5. DETAILED MICROARCHITECTURE

The goal of PHANTOM's microarchitecture is to make maximum use of the available DRAM bandwidth (e.g. up to 1,024b/cycle on the Convey HC-2ex). This involves accelerating all computation related to Path ORAM such that it can be completely overlapped with the memory accesses.

Figure 8 shows how the microarchitecture components fit together and Figure 9 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).

5.1 Data Structures

The *Position Map* (❶ in Figure 9) 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.

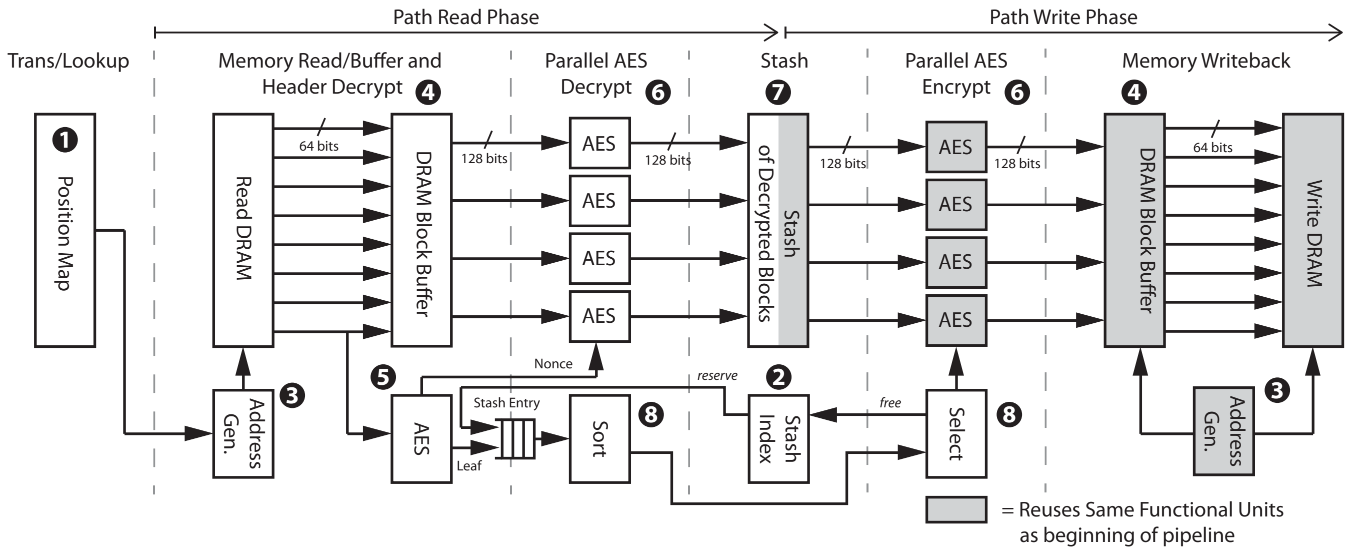

Figure 9: The data path in Phantom resembles a boomerang: Data is read, decrypted and written to the stash. It is then read from the stash, encrypted and written back to memory, using the same functional units.

The *Stash* ❼ buffers ORAM blocks in plaintext. Its size is chosen to make the overflow probability small (Section 3.2). 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 5.3). 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 . Querying the Stash Index and the Position Map adds a single cycle latency to each request. Position Maps grow linearly with ORAM sizes and may have to be pipelined to keep the critical timing path low (up to 4-deep for the position map of a 17-level tree, representing an additional 4 cycles of latency). Updating the stash index requires 2 cycles, but these are overlapped with the path reads/writes.

5.2 DRAM Interface

Phantom's DRAM Interface takes a leaf ID and fetches an entire path from the untrusted DRAM into the trusted memory on the FPGA.

Memory Request Generation: The Convey platform features 1,024 DRAM banks (64 for each of its 16 memory channels). We cannot place ORAM blocks sequentially in the memory address space if we want to fetch data at (close to) peak bandwidth. Instead, we place each word such that the DRAM banks' latency can be hidden behind successive 64 bit words being read from different DRAM banks.

We stripe each ORAM block across the 1,024 banks such that every memory controller accesses all its banks in sequence, reading a 64 bit word from each bank, 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.

DRAM Buffer: As encrypted data arrives from memory, it is put into the *DRAM Buffer* 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 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 thus isolates the internals of Phantom from these timing variations. As long as it 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.

The DRAM Buffer is implemented using 16 independent columns. 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 rest of the system that it is ready to be consumed.

5.3 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 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 stored in its first 128 bits. This is followed by the block's (encrypted) header which includes the block's address 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.

Decryption is overlapped with the rest of the algorithm by adding a forwarding path ❺ from the first two memory channels. It 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.

Further, to avoid wasting oblivious storage, we store the nonce in the block header itself (since more than 80 out of 128 header bits are free—even for large configurations). 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.

Decrypting the header in advance also allows Phantom to insert a block into its proper place in the reorder heap (Section 5.4) as the rest of the block is being read in. Phantom reserves 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.

The choice 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 nonce, plain text and counter as input, and output the cipher text 11 cycles later. We also assume an on-chip pseudo-random number generator. 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 [18] would require 50% of the logic). 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).

5.4 Heap-based Reordering

As described in Section 3, 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 to choose a candidate block for each slot. However, this search would take a larger number of cycles than the write-back of the block to memory would take, making Phantom computationally bound rather than memory bound (especially on an FPGA).

For this to be avoided, Phantom has $max\_time = (block\text{-}size/bits\_per\_cycle)$ cycles to pick a block to write back. In our prototype with 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 – roughly 1,200 additional cycles for a 17-level ORAM. To avoid these stalls, we need a different approach to overlap reordering with memory accesses.

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), 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.

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) .

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 to be linear in the stash size and larger than max_time (i.e. 32 cycles).

We therefore designed Phantom to use a min-heap instead. Rather than performing all the sorting either during the write phase or during the read phase, we insert the blocks into a heap as they are read in, and remove them from the heap as we write them back. This takes logarithmic time both during reading 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 path. We therefore added a queue (which holds as many entries as the stash) to buffer 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 sorting by the time the last block has finished reading (a proof can be found in Appendix A).

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

5.5 Control Logic

Phantom's control logic coordinates the movement of data through the ORAM controller, namely from the DRAM buffer through the AES units to the stash (and vice versa for the write-back), maintaining the invariant that data takes constant number of cycles to flow down 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

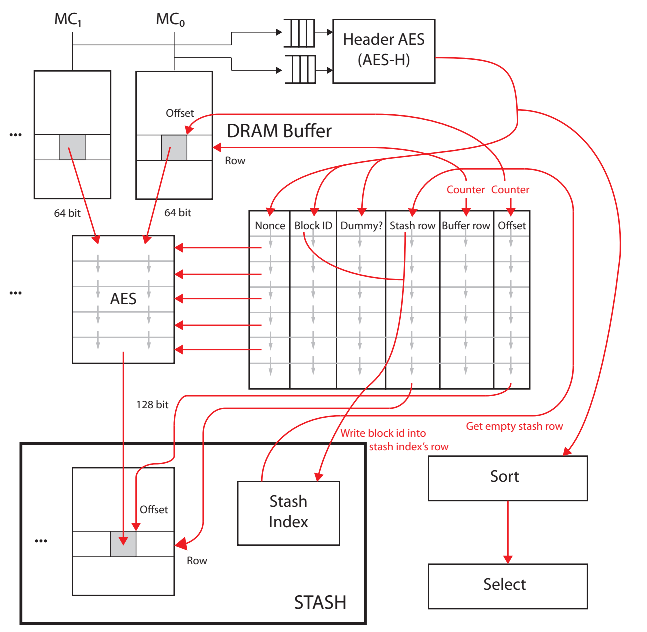

Figure 10: The Control Logic The arrows describe the flow of data through the logic.

pipeline holds a *descriptor* for the data in the corresponding stage of the AES units (Figure 10), which is a tuple comprising an ORAM header, a row in the DRAM buffer, a stage in the AES pipeline, and an entry in the stash.

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).

The write phase uses the same pipeline and AES units, but with different descriptors. As data is moved out from the stash to the DRAM buffer, the control logic queries the select stage (Section 5.4) to determine whether the next block to be written is a dummy, and if not, which entry from the stash to write back. Further, all block headers are updated with a new nonce.

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. 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).

5.6 FPGA-specific Challenges

PHANTOM has to overcome several challenges unique to an implementation on an FPGA:

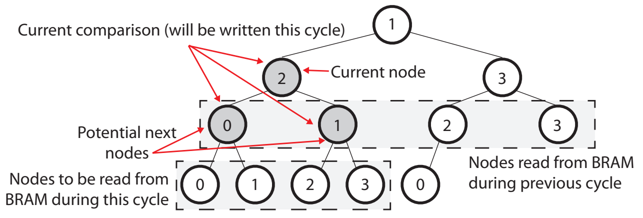
Current comparison (will be written this cycle)
1
2
Current node
3
Potential next nodes
0
1
2
3
Nodes to be read from BRAM during this cycle
0
1
2
3
0
Nodes read from BRAM during previous cycle

Figure 11: Min-heap on the FPGA's on-chip RAM. The numbers represent the RAM each node is stored in. The four second-degree children of each node are put into four separate RAMs in order to read them in parallel.

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. Furthermore, we perform reads and updates concurrently every cycle.

Though 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 11). 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 goes.

The limit of one memory read and one write port also complicates the stash's design. While there are four units that access each stash column (read for encryption, write for decryption and reads/writes by the secure CPU), these operations are scheduled such that only two ports are required at the same time (reading the response data to the CPU is overlapped with decryption, writing with encryption).

5.7 Integrating ORAM with a CPU

Our ORAM controller is integrated with an in-order RISC processor implementing the RISC-V instruction set [37]. The processor (like our ORAM controller) is implemented using Chisel [5], a new hardware construction language embedded in Scala. The CPU features a full implementation of the RISC-V instruction set as well as an uncore with instruction, data and last-level cache (LLC). For our prototype, we use a version 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). Such work is orthogonal to our proposed ORAM controller (for our performance evaluation, we simulate a larger CPU with reasonable cache sizes to determine results for a realistic deployment).

The CPU is integrated with PHANTOM by translating memory requests from the LLC into ORAM requests, buffering 8 previously used ORAM blocks to reduce the miss rate. We use the CLOCK algorithm to determine which ORAM block to evict from the buffer. Furthermore, translating memory addresses into ORAM block addresses is non-trivial since it requires division by a non-power of two (due to the 128 bit used by the block header). Our current prototype uses a Xilinx divider IP core that takes an additional 36 cycles, but custom logic could eventually exploit the fact that this division is by a power-of-two multiple of 31 to reduce that latency to just a few cycles.

5.8 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 – 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.

6. EVALUATION

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 22× over a non-secure access (30× for 128B blocks[3]) for a 64MB ORAM. ORAM access latencies vary from 18us to 30us for ORAMs of effective size 64MB to 4GB and a block size of 4kB. A non-ORAM access takes 812ns for a 4kB block and 592ns for a 128B block. 2) PHANTOM utilizes 93% of the theoretical peak bandwidth of the memory channel, validating our memory layout and micro-architectural optimizations. 3) PHANTOM requires less than 2% of the logic of the FPGA, leaving space to implement other accelerators or a CPU. PHANTOM's prototype implements ORAMs of up to 1GB on a single FPGA (before on-chip memory is exhausted) and 2GB or larger using multiple FPGAs. 4) The overall program performance depends upon access patterns and working set size. Different SQLite queries on a population census workload shows overheads from 20% to 5×.

6.1 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).

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.

[3]Since the granularity of ORAM accesses is much larger than a cache line, we compare to 128B reads to estimate the worst-case overhead where only a fraction of data is used.

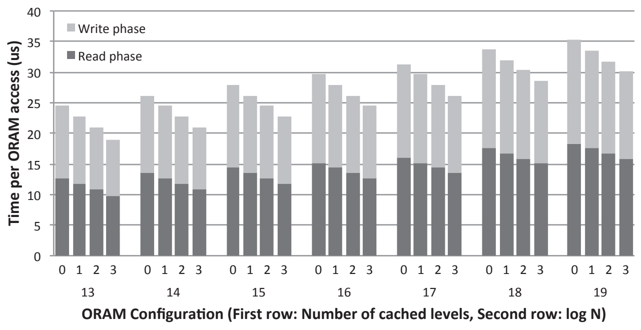

Figure 12: Average time per ORAM access for different configurations on hardware. 1M accesses each, block size 4KB, 18 & 19 levels are split across 3 FPGAs.

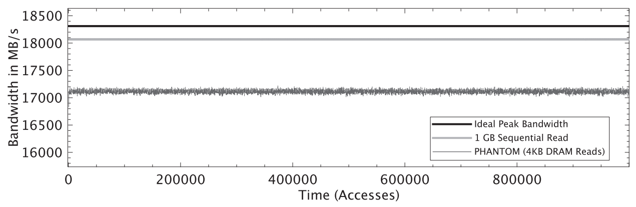

Figure 13: DRAM utilization over time.

Figure 12 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 (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 (for reads) – this is important since a client 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 16 us 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, this represents 12× to 27× overhead. An ORAM 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, 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.

DRAM Utilization. Figure 13 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

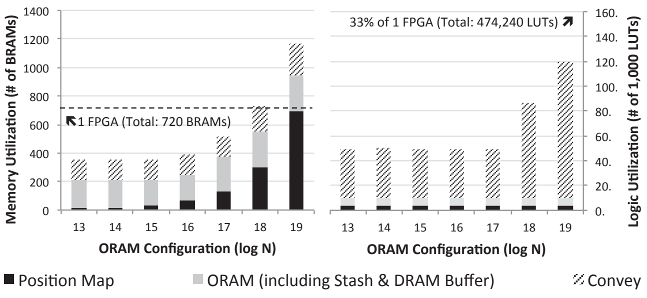

Figure 14: FPGA resource utilization. The design uses 30-52% (less than 2%) of available memory (logic); up to 74% (10%) including Convey's interface. It fits onto a single FPGA for up to 17 levels (1GB). For larger sizes, the position map is stored remotely on other FPGAs.

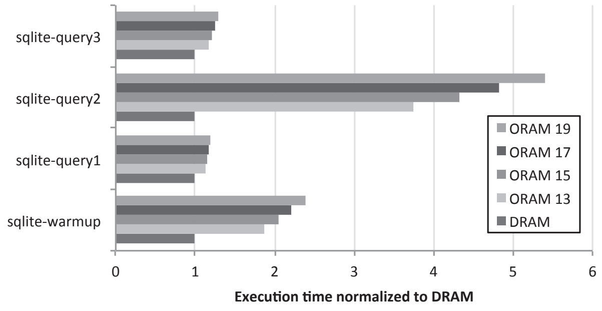

Figure 15: SQLite Performance Results. We simulated the performance of sqlite running with and without PHANTOM on a timing model of a processor on the same FPGA. We assume a 1MB L2 cache, 16KB icache, 32KB dcache and an extra buffer for 8 ORAM blocks (32KB).

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[4]. 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.

6.2 FPGA Resource Usage

Figure 14 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 on-chip RAMs (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 x86 core.

This breakdown shows that the biggest contributors to memory usage are the position map and the stash. To support ORAM sizes larger than 1 GB, we therefore move the lookup table to another FPGA and communicate through (encrypted) messages over Convey's inter-FPGA communication facilities (Section 5.8).

These results do not include the resources that would be consumed by real AES crypto hardware. There exist optimized AES processor designs [18] 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).

6.3 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.

As an end-to-end example, we used our real RISC processor to run several SQLite queries on the 7.5MB census database from Figure 1 on our real hardware. Due to the processor's extremely small cache sizes, a very large percentage of memory accesses are cache misses (we ran the same workloads on a RISC-V simulator and found 7.7% dcache misses, and 55.5% LLC misses). As a result, we observed slow-downs 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-downs compared to a baseline without ORAM).

To investigate the effect on applications in the presence of realistic cache sizes (namely a 16KB cache, 32KB cache and 1MB LLC cache), we fed the same application traces into a timing model derived from our real processor 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 DRAM configurations. We also assumed that the processor waits for a full ORAM request to finish and additionally evict a previous block, essentially causing an up to 4× longer stall than a real execution. At the same time, we allow the baseline to use the full Convey memory system.

Despite these penalties, 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 15 shows our results for applying our timing model to several SQLite queries on the 7.5MB census database from Figure 1. `sqlite-warmup` and `sqlite-query1` are a `JOIN` operation between two large tables. The difference in execution times show the effect 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 poorer 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 also like to point out that the baseline would also benefit from pipelining requests, which we do not account for).

In summary, our evaluation shows that the overheads exposed by PHANTOM are indeed reasonable for real-world workloads, especially given the fundamental cost of obliviousness is a bandwidth overhead of greater than 100×. Note

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

that these numbers depend upon 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.

7. RELATED WORK

In the introduction, we highlighted representative prior work on secure processors. Here we focus on work on obliviousness, especially in the context of secure processors. Oblivious RAM (ORAM) is an algorithmic construct for hiding memory access patterns and was first proposed by Goldreich and Ostrovsky [12]. ORAM hides only memory access patterns, but does not hide the timing or the total number of memory accesses – complementary techniques can be used to secure timing and termination channels [2, 3, 7, 16, 41].

Recent research on ORAM [6, 10, 12–14, 21, 22, 24, 28, 31, 39, 40] shows that it is possible to achieve obliviousness with $O(\text{polylog}(N))$ overhead, i.e., every memory access translates into $O(\text{polylog}(N))$ seemingly random accesses.

The above algorithms have been demonstrated for disk accesses in datacenters [22, 30, 40], but are impractical to implement in a hardware memory controller since they are complex and have high constant factors. Some prior work (HIDE [42]) has looked at probabilistic solutions in hardware that do not provide complete obliviousness but have far lower overheads. Where strong obliviousness guarantees are not required, HIDE might be a better fit.

In this project, we build upon Path ORAM [32], an ORAM algorithm that is much simpler than prior ORAM algorithms. Path ORAM has also been adopted by ASCEND [9, 26], a proposed secure processor with an oblivious memory interface. While similar on the surface, we believe our work to be largely complementary to ASCEND. The authors apply optimizations such as hierarchical ORAM, background eviction, and integrity verification to Path ORAM. These optimizations can be applied on top of Phantom as well, while our optimizations are directed towards the hardware architecture for Path ORAM. Using simulations, they predict that a SPEC 2006 int benchmark subset can execute with a 3× (geometric mean) performance overhead over native. Note that with an ORAM latency of more than 2000 cycles, the low overhead can be attributed as much to the SPEC benchmarks' working set fitting in the cache (similar to our results for SQLite).

8. CONCLUSION

We present Phantom, a practical oblivious memory system that achieves high performance by exploiting memory parallelism and using an architecture that scales to wide memory channels. Importantly for adoption, implementing Phantom prototype on a Convey FPGA machine enables obliviousness to be available today. In the future, Phantom can expose a memory composed of DRAM, encrypted DRAM, and ORAM banks to software and thus open the door to compiler analyses that improve performance without compromising obliviousness.

Acknowledgements

We would like to thank John Lazzaro, John Wawrzynek and Brian Zimmer (for CS250 instruction, Fall'11), Yunsup Lee and Andrew Waterman (for RISC-V support), Derek Chiou and Walid Najjar (for Convey hardware), and Glen Edwards and George Vandegrift (for technical advice and support). Last but not least, we thank the anonymous reviewers for their comments.

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.

9. REFERENCES

[1] "PrivateCore," http://www.privatecore.com/.
[2] J. Agat, "Transforming out Timing Leaks," in *POPL*, 2000.
[3] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *CCS*, 2010.
[4] A. Aviram, S. Hu, B. Ford, and R. Gummadi, "Determinating Timing Channels in Compute Clouds," in *CCSW*, 2010.
[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing Hardware in a Scala Embedded Language," in *DAC*, 2012.
[6] K.-M. Chung, Z. Liu, and R. Pass, "Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead," http://arxiv.org/abs/1307.3699, 2013.
[7] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors," in *SP*, 2009.
[8] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic Shared Memory Multiprocessing," in *ASPLOS*, 2009.
[9] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A Secure Processor Architecture for Encrypted Computation on Untrusted Programs," in *STC*, 2012.
[10] C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing oram and using it efficiently for secure computation," in *PETS*, 2013.
[11] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," in *STOC*, 1987.
[12] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. ACM*, 1996.
[13] M. T. Goodrich and M. Mitzenmacher, "Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation," in *ICALP*, 2011.
[14] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving Group Data Access via Stateless Oblivious RAM Simulation," in *SODA*, 2012.
[15] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin, "OS-Sommelier: Memory-only Operating System Fingerprinting in the Cloud," in *SoCC*, 2012.
[16] A. Haeberlen, B. C. Pierce, and A. Narayan, "Differential Privacy Under Fire," in *USENIX Security*, 2011.
[17] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold-boot Attacks on Encryption Keys," *Commun. ACM*, vol. 52, no. 5, 2009.

[18] A. Hodjat and I. Verbauwhede, “A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA,” in *FCCM*, 2004.

[19] A. Huang, “Keeping Secrets in Hardware: The Microsoft Xbox Case Study,” in *CHES*, 2002.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: Formal Verification of an OS Kernel,” in *SOSP*, 2009.

[21] E. Kushilevitz, S. Lu, and R. Ostrovsky, “On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme,” in *SODA*, 2012.

[22] J. R. Lorch and B. Parno, “Shroud: Ensuring Private Access to Large-Scale Data in the Data Center,” in *FAST*, 2013.

[23] R. Martin, J. Demme, and S. Sethumadhavan, “TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks,” in *ISCA*, 2012.

[24] R. Ostrovsky and V. Shoup, “Private Information Storage (Extended Abstract),” in *STOC*, 1997.

[25] L. Ren, C. Fletcher, X. Yu, M. van Dijk, and S. Devadas, “Integrity verification for path oblivious-ram,” in *HPEC*, 2013.

[26] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas, “Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors,” in *ISCA*, 2013.

[27] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, “Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly,” in *MICRO*, 2007.

[28] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, “Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost,” in *ASIACRYPT*, 2011.

[29] S. W. Smith, “Outbound Authentication for Programmable Secure Coprocessors,” in *ESORICS*, 2002.

[30] E. Stefanov and E. Shi, “Oblivistore: High performance oblivious cloud storage,” in *S & P*, 2013.

[31] E. Stefanov, E. Shi, and D. Song, “Towards Practical Oblivious RAM,” in *NDSS*, 2012.

[32] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, “Path O-RAM: An Extremely Simple Oblivious RAM Protocol,” in *CCS*, 2013.

[33] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing,” in *ICS*, 2003.

[34] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural Support for Copy and Tamper Resistant Software,” *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 5, pp. 168–177, 2000.

[35] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, “Complete Information Flow Tracking from the Gates up,” in *ASPLOS*, 2009.

[36] A. Waksman and S. Sethumadhavan, “Silencing Hardware Backdoors,” in *SP*, 2011.

[37] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA,” EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62, May 2011.

[38] P. Williams and R. Sion, “Round-Optimal Access Privacy on Outsourced Storage,” in *CCS*, 2012.

[39] P. Williams, R. Sion, and B. Carbunar, “Building castles out of mud: practical access pattern privacy and correctness on untrusted storage,” in *CCS*, 2008.

[40] P. Williams, R. Sion, and A. Tomescu, “PrivateFS: A Parallel Oblivious File System,” in *CCS*, 2012.

[41] D. Zhang, A. Askarov, and A. C. Myers, “Predictive Mitigation of Timing Channels in Interactive Systems,” in *CCS*, 2011.

[42] X. Zhuang, T. Zhang, and S. Pande, “HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus,” in *ASPLOS*, 2004.

APPENDIX

A. PROOF OF OVERLAPPING HEAPIFY.

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 but the ones on the last level, where k is the distance of the node to the leaves. Hence *heapify* takes

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

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 as described in Section 5.4. Note that this bound is not very tight (for $d \leq 8$, $t_h = 374$) and sorting can be further optimized.