Tiny ORAM: A Low-Latency, Low-Area Hardware ORAM Controller

Anonymized for submission

***Abstract*—We build and evaluate *Tiny ORAM*, an Oblivious RAM prototype on FPGA. Oblivious RAM is a cryptographic primitive that *completely* obfuscates an application's data, access pattern, and read/write behavior to/from external memory (such as DRAM or disk).**

Tiny ORAM makes two main contributions. First, by removing an algorithmic bottleneck in prior work, Tiny ORAM is the first hardware ORAM design to support arbitrary block sizes (e.g., 64 Bytes to 4096 Bytes). With a 64 Byte block size, Tiny ORAM can finish an access in $1.4\mu s$, over $40\times$ faster than prior work. Second, through novel algorithmic and engineering-level optimizations, Tiny ORAM reduces the number of symmetric encryption operations by $\sim 3\times$ compared to prior work. Tiny ORAM is also the first design to implement and report real numbers for the cost of symmetric encryption in hardware ORAM constructions. Putting it together, Tiny ORAM requires $5\%/13\%$ of the FPGA logic/memory, including the cost of encryption.

I. Introduction

With cloud computing becoming increasingly popular, privacy of users' sensitive data has become a huge concern in computation outsourcing. Ideally, users would like to "throw their encrypted data over the wall" to a cloud service that performs computation on that data, but cannot obtain any information from within that data. It is well known, however, that encryption is not sufficient to enforce privacy in this environment, because a program's memory access pattern reveals a large percentage of its behavior [27] or the encrypted data it is computing upon [14], [16].

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [11], [12], is a cryptographic primitive that completely eliminates the information leakage from a programs memory access trace, i.e. the sequence of memory accesses. ORAM is made up of trusted client logic (who runs the ORAM algorithm and maintains some trusted state) that interacts with an untrusted storage provider. Conceptually, ORAM blocks information leakage by maintaining all memory contents encrypted and memory locations randomly shuffled. On each access, memory is read and then reshuffled. Under ORAM, any memory access pattern is computationally indistinguishable from any other access pattern of the same length. Since the original proposal, there has been significant work that has resulted in more efficient and cryptographically-secure ORAM schemes [18], [17], [6], [4], [13], [15], [24], [21], [23]. The cost for ORAM security is performance: to read/write a block (the atomic unit of data a client may request), ORAM moves a logarithmic number of blocks over the chip pins.

An important use case for ORAM is in *trusted hardware* [16], [8], [22], [23], [20], [7]. Figure 1 shows an example target cloud configuration, where a client communicates with trusted secure processor in the cloud, which is attached to untrusted external memory such as disk, DRAM or flash. In

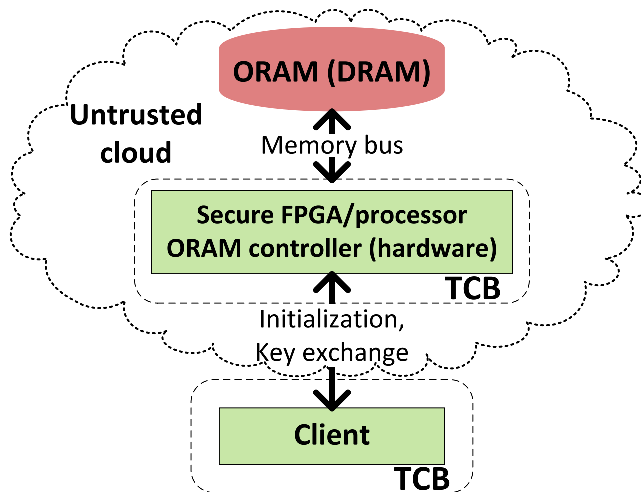

Fig. 1. ORAM in the secure FPGA/processor setting.

this configuration, the ORAM client logic (called the ORAM controller) is typically implemented in hardware and, thus, needs to be simple. Therefore, most secure processor proposals [16], [8], [23], [20], [7] have adopted Path ORAM because of its small client storage and simplicity. For the rest of this paper, we assume the untrusted memory is implemented in DRAM and that the adversary is passive: it observes but does not tamper with the memory. As in prior ORAM work, we don't consider leakage over timing, power, RF or other side channels.

A. Opportunities and Challenges for ORAM on FPGA

Because of their programmability and high performance, FPGAs have attracted significant attention as accelerators for cryptographic primitives. FPGAs have a key advantage over ASICs, namely programmability: it is easy to patch FPGA designs in the event that a security bug is discovered or a cryptographic primitive is rendered obsolete [25]. Partly for this reason, major FPGA vendors have added dedicated cryptographic key management logic into their mainstream FPGAs to support secure sessions and bitstream reconfiguration [26], [5]. At the same time, FPGAs have a key advantage over software on performance, due to hardware-level optimization. For example an FPGA implementation of Keccak [3], chosen by NIST as SHA-3, has been shown to outperform an optimized software implementation by over $5\times$ [9], [2].

Following this trend, we argue that FPGAs are also very appealing as target platforms for ORAM. Beyond the points we have already made, modern FPGAs are often designed

with dedicated memory controllers to achieve high memory bandwidth. For example, the Xilinx Virtex-7 VC709 board can achieve a memory bandwidth of 25 GigaByte/second using dual DDR3 channels, which is similar to midrange desktop systems like the Intel Core i7 processor family. Meanwhile, FPGA designs typically run in 100-300 MHz range. This fabric/memory clock disparity means the performance cost of ORAM is significantly dampened on FPGA. Indeed, our Tiny ORAM's latency to access a block is less than 200 cycles, compared to over 1000 cycles estimated on ASIC [20], [7].

The challenge in designing ORAM for FPGA is exactly how to saturate this plentiful memory bandwidth. For instance, the Virtex-7 VC709's 25 GigaByte/second bandwidth translates to 1024 bits/FPGA cycle, forcing the ORAM algorithm and implementation to have 1024 bits/FPGA cycle throughput. Recently Maas et al. [16] implemented *Phantom*, the first hardware ORAM prototype for the trusted hardware setting. Their design identifies several performance and resource bottlenecks which we address in this paper.

The first bottleneck occurs when the application block size is small. Phantom was parameterized for 4-KiloByte blocks and, even after optimizations to decrease minimum block size, cannot support less than 704-1408 Byte blocks without introducing pipeline stalls (§ III). This minimum block size grows with FPGA memory bandwidth. While benefiting applications with good data locality, a large block size (like 4KBytes in Phantom) severely hurt performance for applications with erratic data locality[1]. The first goal in this paper is to develop schemes that flexibly support any block size (e.g., we evaluate 64-Byte blocks) without incurring performance loss.

The second bottleneck is that to keep up with FPGAs' large memory bandwidth, an ORAM controller requires many encryption units, imposing large area overheads. This is because in prior art ORAM algorithms, all blocks transferred must be decrypted/re-encrypted, so encryption bandwidth must scale with memory bandwidth. Phantom projects that AES units alone would take $\sim$ 50% of the logic of a state-of-the-art FPGA device. The second goal in this paper is to develop new ORAM schemes that reduce the required encryption bandwidth, and to carefully engineer the system to save hardware area.

B. Contributions

In this paper, we present *Tiny ORAM*, a complete hardware ORAM controller prototype implemented on an FPGA. Through novel algorithmic improvements and careful hardware design, Tiny ORAM makes two major contributions: (1) it enables configuration with small blocks, achieving low latency, and (2) it has an extremely small hardware area footprint. We achieve these goals through two novel techniques:

Bit-based stash management to enable small blocks size (§ III). We develop a new stash management scheme using efficient bit tricks that, when implemented in hardware, removes the block size bottleneck in the Phantom design [16]. In particular, our scheme can support any reasonable block size (e.g., from 64-4096 Bytes) without sacrificing system performance. With a 64 Byte block size, Tiny ORAM improves the access latency by $\geq 40\times$ in the best case compared to Phantom.

[1]Most modern processors have a 64-Byte cache block size for this reason.

TABLE I
ORAM PARAMETERS AND NOTATIONS.

Notation	Meaning
L	Depth of Path ORAM tree
Z	Data blocks per ORAM tree bucket
N	Number of real data blocks in tree
B	Data block size (in bits)
C	Stash capacity (in blocks, excluding transient storage)
K	Session key (controlled by trusted processor)
$\mathcal{P}(l)$	Path to leaf l in ORAM tree
$\mathcal{P}(l)[i]$	i-th bucket on Path $\mathcal{P}(l)$
RAW ORAM (§ IV) only	
A	The number of RO accesses per RW access

RAW ORAM Path Write Predictability to reduce the required encryption engines (§ IV). Inspired by Gentry et al. [10], we propose *RAW ORAM* and *path write predictability (PWP)* to reduce the number of encryption units by $\sim 3\times$ while maintaining comparable bandwidth to basic Path ORAM.

We implement the above ideas in hardware, and evaluate our design for performance and area on a Virtex-7 VC707 FPGA board. With the VC707 board's 12.8 GB/s DRAM bandwidth, Tiny ORAM can complete an access for a 64 Byte block in $1.4\mu s$. This design (with encryption units) requires 5% of the FPGA's logic and 13% of its on-chip memory, demonstrating a significantly reduced hardware footprint over existing alternatives. Our design is written entirely in Verilog-2001 with no proprietary components.

We have given an overview of related work and this work in the introduction. The rest of the paper is organized as follows. Section II provides necessary background on ORAM and Path ORAM. Section III introduces the bit-based stash management method, while Section IV introduces RAW ORAM and path write predictability. Section V presents and evaluates our overall Tiny ORAM design, demonstrating improvements over existing ORAM designs on FPGA.

II. Background

As did Phantom, Tiny ORAM originates from and extends Path ORAM [23]. We now explain Path ORAM in detail. Parameters and notations are summarized in Table I.

A. Basic Path ORAM

Path ORAM organizes untrusted external DRAM as a binary tree which we refer to as the *ORAM tree*. The root node of the ORAM tree is referred to as level 0, and the leaf nodes as level L. We denote each leaf node with a unique leaf label l for $0 \leq l \leq 2^L - 1$. We refer to the list of buckets on the path from the root to leaf l as $\mathcal{P}(l)$.

Each node in the tree is called a *bucket* and can hold up to a small constant number of blocks denoted Z (typically $Z = 4$). We denote the block size in bits as B. In this paper, each block is a processor cache line (so we correspondingly set $B = 512$). Buckets that have less than Z blocks are padded with *dummy blocks*. Each bucket is encrypted using symmetric

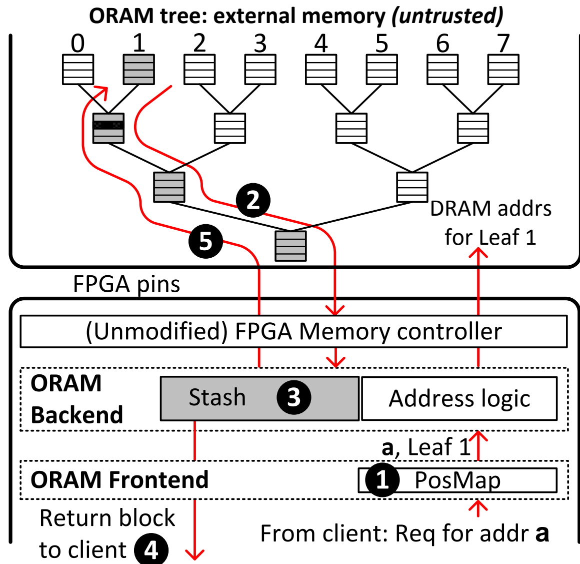

Fig. 2. A Path ORAM of $L = 3$ levels and $Z = 4$ slots per bucket. Suppose block a, shaded black, is mapped to path $l = 1$. At any time, block a can be located in any of the shaded structures (i.e., on path 1 or in the stash).

probabilistic encryption (e.g., AES in counter mode). Thus, an observer cannot distinguish real blocks from dummy blocks.

The Path ORAM controller (trusted hardware) contains a *position map*, a *stash* and associated control logic. The position map (PosMap for short) is a lookup table that associates each data block's logical address with a leaf in the ORAM tree. The stash is a random access memory (e.g., an SRAM) that stores up to a small number of data blocks. The stash capacity is the maximum number of data blocks on a path plus a small number, i.e., $C + (L + 1)Z$. We say that the stash has overflowed if, at the beginning of an ORAM access, the number of blocks in the stash is $> C$. To achieve negligible stash overflow probability for real security parameters, $C = 50$ to 100 is typical. Together, the PosMap and stash make up Path ORAM's client storage.

Path ORAM Invariant. At any time, each data block in Path ORAM is mapped to a random leaf via the PosMap. Path ORAM maintains the following invariant: *If a block is mapped to leaf l, then it must be either in some bucket on path l or in the stash.* Blocks are stored in the stash or ORAM tree along with their current leaf and block address.

To make a request for a block with address a (block a for short), the Last Level Cache (LLC) or FPGA user design calls the ORAM controller via accessORAM(a, op, d'), where op is either read or write and d' is the new data is op =write. The steps are also shown in Figure 2.

1) Look up PosMap with a, yielding the corresponding leaf label l. Randomly generate a new leaf l' and update the PosMap for a with l'.
2) Read and decrypt all the blocks along path l. Add all the real blocks to the stash (dummies are discarded). Due to the Path ORAM invariant, block a must be in the stash at this point.
3) Update block a in the stash to have leaf l'.
4) If op = read, return block a to the LLC. If op = write, replace the contents of block a with data d'.
5) Evict and encrypt as many blocks as possible from the stash to path l in the ORAM tree (to keep the stash occupancy low) while keeping the invariant. Fill any remaining space on the path with encrypted dummy blocks.

Note that all DRAM read/write steps are performed through an unmodified FPGA memory controller (e.g., Xilinx MIG).

To simplify the presentation, we refer to Step 1 (the PosMap lookup) as the Frontend(a), or Frontend, and Steps 2-5 as the Backend(a, l, l', op, d'), or Backend. This work optimizes the Backend only, but we demonstrate a complete system with a working Frontend in our evaluation for completeness.

Stash eviction in hardware. In the above algorithm, performing Step 5 efficiently is a big challenge for hardware designs [16]. One contribution in this paper is a simple mechanism that solves the problem for any reasonable block size or memory bandwidth (§ III).

Bucket header. Implicit in the Path ORAM algorithm, each block is stored in the stash and ORAM tree alongside its program address and current leaf. Besides Z data blocks, each bucket also stores the *bucket header*, which includes addresses/leaves of the blocks and an initialization vector used for symmetric encryption. Dummy blocks have a special program address $\perp$. All bucket fields are shown in Figure 3 for the parameterization we evaluate at the end of the paper.

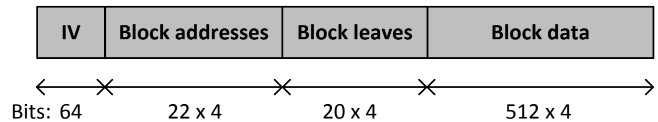

Fig. 3. Bucket fields. We show field sizes for the parameterization we evaluate in § V, namely using a 1 GB DRAM DIMM. For this setting, we set $L = 20$, $Z = 4$ (as suggested by prior work [16], [23]) and fill the ORAM tree until 50% of available slots are used for data blocks (following [20]). IV refers to the initialization vector used for encryption.

Path ORAM Security. The intuition for Path ORAM's security is that every PosMap lookup (Step 1) will yield a fresh random leaf that has never been revealed before. This makes the sequence of ORAM tree paths accessed and the program address trace independent. Further, probabilistic encryption ensures computational indistinguishability regardless of whicheverblock is read on the path. We use AES counter mode with counter width being 64 bits (see Figure 3). Each counter increments once per ORAM access; i.e., no counter will overflow even after 100s of years of constant accesses. For stash overflow probability, we set bucket size $Z = 4$ following [23], [16], and provision a stash capacity of $C = 78$, which gives a stash overflow probability of 2^{-80} (using the methodology from [16]).

B. Recursive Path ORAM

In basic Path ORAM, the number of entries in the PosMap (§ II-A) scales linearly with the number of data blocks in the ORAM. This results in a significant amount of on-chip storage — indeed, Phantom [16] required multiple FPGA's just

to store the PosMap for sufficiently large ORAMs. Recursive ORAM was first proposed by Shi et al. [21] to solve this problem and has been studied through simulation in trusted hardware proposals [8], [20]. The idea is to store the PosMap in a separate ORAM, and store the new ORAM's (smaller) PosMap on-chip.

We refer to the original ORAM as the *Data ORAM*, the second ORAM a *PosMap* ORAM. Accessing data block a in the recursive construction involves two ORAM accesses. The first is to the PosMap ORAM, to retrieve the leaf label of block a, and the second is to Data ORAM for block a. Of course, the new on-chip PosMap might still be too large. In that case, additional PosMap ORAMs may be added to further shrink the on-chip PosMap.

[7] demonstrates how to reduce the bandwidth cost of recursion by 95% while increasing the area of a baseline Recursive design by $\sim 5\%$. Similarly to our work, [7] breaks ORAM into a Frontend and a Backend, with the same interfaces, and implements recursion entirely as changes to the Frontend. As we mentioned earlier, this paper's focus is to optimize the Backend only. Thus, our optimizations naturally extend to the recursive setting of [7] and we evaluate a combined system, using the optimized Frontend of [7], for completeness.

III. Stash Management

As mentioned in § II-A, evicting blocks from the stash is a challenge for efficient Path ORAM hardware designs. Conceptually, we need to push every block in the stash to the deepest possible bucket on path $\mathcal{P}(l)$ while maintaining the Path ORAM invariant. A naïve implementation is, for each slot on the path, to scan the stash sequentially and choose at the end of each scan which block should be pushed to that slot [16]. This design takes $O(C+L*Z)$ cycles per block and must run $L*Z$ times per ORAM access. As pointed out by Phantom, this method causes serious hardware performance bottlenecks. To lessen this bottleneck, Phantom proposes an FPGA-optimized *heap sort* algorithm to manage the stash. The idea is to write blocks to the stash in eviction-sorted order, so that each block eviction can be done in a single cycle. With this scheme, writing each block to the stash requires $O(\log(C+L*Z))$ cycles per block.

Unfortunately, Phantom's heap-sort-based stash management algorithm still implies a performance bottleneck for small block sizes. With Phantom's parameters, the pipelined heap sort design takes 11 cycles to evict a block (see Appendix A of [16]). Assuming a memory bandwidth between 512-1024 bits/cycle for modern FPGAs, this constrains the block size B to be at least 11 times 512-1024 bits, to hide the heap-sort latency. In other words, when the block size is <704 to 1408 Bytes, the ORAM controller's performance bottleneck is stash management logic, rather than memory bandwidth.

We now detail a new and simple stash eviction algorithm based on bit-level hardware tricks that takes a single cycle to evict a block and can be implemented efficiently in FPGA logic. This *eliminates* the above performance overhead for any practical block size and memory bandwidth.

A. PushToLeaf With Bit Hacks

Our proposal, the PushToLeaf() routine, is shown in Algorithm 1. PushToLeaf(Stash, l) is run at the beginning of Step 5 during each ORAM access (§ II-A) and yields an array of blocks, denoted $\mathcal{S}$, in the order that they should be written back to $\mathcal{P}(l)$ of the ORAM tree. $\mathcal{S}[i]$ represents the block to be written back to the i-th position on $\mathcal{P}(l)$, of which there are $(L+1)*Z$. Index 0 is in the root bucket. At a high level, our PushToLeaf() routine is a hardware circuit that sequentially, for each block in the stash, pushes that block (PushBack()) as far towards to the leaf bucket along $\mathcal{P}(l)$ as possible using combinational logic.

Suppose l is the current leaf being accessed. We represent leaves as L-bit words which are read right-to-left: the i-th bit indicates whether the i-th bucket's child is the left child (0) or right child (1). On Line 3, we initialize the contents of $\mathcal{S}$ to all dummy blocks, represented by $\perp$. Occupied is an $L+1$ entry memory that records the number of real blocks that have been pushed back to each bucket so far.

Algorithm 1 Bit operation-based stash scan. 2C stands for two's complement arithmetic.

```
 1: Inputs: The current leaf l being accessed
 2: function PushToLeaf(Stash, l)
 3:     S ← {⊥ for i = 0,...,(L+1)*Z−1}
 4:     Occupied ← {0 for i = 0,...,L}
 5:     for i ← 0 to C+L*Z−1 do
 6:         (a, l_i, D) ← Stash[i]          ▷ Leaf assigned to i-th block
 7:         level ← PushBack(l, l_i)
 8:         if a ≠ ⊥ and level > −1 then
 9:             offset ← level*Z + Occupied[level]
10:             S[offset] ← (a, l_i, D)
11:             Occupied[level] ← Occupied[level] + 1
12: function PushBack(l, l′)
13:     t1 ← (l ⊕ l′)||0                      ▷ Bitwise XOR
14:     t2 ← t1 & −t1              ▷ Bitwise AND, 2C negation
15:     t3 ← t2 − 1                         ▷ 2C subtraction
16:     full ← {(Occupied[i] =? Z) for i = 0 to L}
17:     t4 ← t3 & ∼full              ▷ Bitwise AND/negation
18:     t5 ← reverse(t4)                  ▷ Bitwise reverse
19:     t6 ← t5 & −t5
20:     t7 ← reverse(t6)
21:     if t7 =? 0 then
22:         return −1                  ▷ Block is stuck in stash
23:     return log2(t7)            ▷ Note: t7 must be one-hot
```

We now explain the PushBack() routine in detail. Line 13 first concatenates 0 to both l and l' and XORs these vectors together. t_1 now represents in which levels the paths $P(l)$ and $P(l')$ diverge. Line 14 then clears all remaining bits *except* for the *right-most set bit*. t_2 is now called "one-hot" (meaning it contains exactly 1 set bit) and its set bit indicates the *first* level where $P(l)$ and $P(l')$ diverge. Line 15 converts t_2 to a vector of the form $000\ldots111$, where set bits indicate which levels the block *can* be pushed back to. Line 17 further excludes buckets that already contain Z blocks (i.e., from previous calls to PushBack()). Finally, Lines 18-20 turns all current bits off except for the *left-most set bit*, which indicates the highest level towards the leaves that the block can be pushed back to.

Since our PushToLeaf() routine does not assume any order of blocks in the stash (i.e., is unsorted), we can also add a block to the stash in 1 cycle per block. In our current design, we manage the stash as a simple linked-list and simply add a node to the list to insert a block.

B. Hardware Implementation and Pipelining

Algorithm 1 runs $O(C + L * Z)$ iterations of PushBack() per ORAM access. $O(C)$ iterations are spent scanning blocks that may have been in the stash at the beginning of the access. The remaining iterations process blocks on the current path. In hardware, we pipeline Algorithm 1 in three respects to hide this $O(C + L * Z)$ operation.

First, the PushBack() circuit itself is pipelined to have (amortized) throughput of 1 block / cycle. PushBack() itself synthesizes to simple combinational logic where the most expensive operation is two's complement arithmetic of $(L+1)$-bit words (which is also cheap due to optimized FPGA carry chains). Implemented in hardware, reverse() costs no additional logic and the other bit operations (including $\log_2(x)$ when x is one-hot) synthesize to LUTs. To meet our FPGA's clock frequency, we had to add 2 pipeline stages after Lines 14 and 15. Thus, performing C iterations of PushBack() requires $C + 2$ cycles.

Second, as soon as an ORAM access starts and the leaf being read is presented to the Backend (i.e., concurrent with Step 2 in § II-A), blocks already in the stash are sent to the PushBack() circuit "in the background". Following the previous paragraph, $C + 2$ is the number of cycles it takes to perform the background scan in the worst case.

Third, after cycle $C + 2$, we send each block read on the path to the PushBack() circuit *as soon as it arrives from DRAM*. Since a new block can be processed each cycle, we are guaranteed to be able to start writing back blocks to the ORAM tree several cycles after the *last block* read from DRAM is processed by PushBack().

C. Data-Independent Timing for Security

Following Phantom, we design each Tiny ORAM access to have data-independent timing: the timing of all observable behaviors of Tiny ORAM (read/write DRAM, return data to requester, etc.) can be predicted based on public information. Let C' be the number of cycles between when an ORAM access starts and when the first block from DRAM arrives at the stash. We design Tiny ORAM to always incur a $\max(0, C + 2 - C')$ worst-case stall to avoid data-dependent timing variations. This cost is once *per ORAM access* and in the tens of cycles in practice. (We will give a concrete example assuming our evaluation parameters in § V-B.)

IV. Minimizing Design Area

Another serious problem for ORAM design is the area needed for symmetric encryption units. Recall from § II-A that all data read and written by ORAM must get decrypted and re-encrypted to preserve privacy. Encryption bandwidth hence scales with memory bandwidth and quickly becomes the area bottleneck. Indeed, Phantom [16] did not implement real encryption units in their design but predicted that encryption would take 50% area of a high-end Virtex 6 FPGA.

To handle this problem, we now propose a new ORAM design, which we call *RAW ORAM*, optimized to minimize symmetric encryption bandwidth at the algorithmic and engineering level. At a high level, our construction is two parts:

Asymmetric operations for inexpensive reads. First, we split PORAMBackend() into two flavors: *read-only* (RO) and *read-write* (RW) accesses. RO accesses perform ORAM requests (i.e., memory read/writes) for the client and RW accesses perform evictions (to empty the stash) in the background. To reduce the number of encryption units needed by ORAM, we optimize RO accesses *to only decrypt the block of interest and a small amount of metadata as opposed to the entire path.* RW accesses require more encryption/decryption, but occur less frequently.

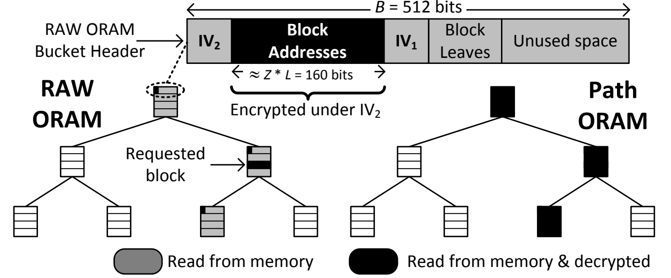

Fig. 4. Data read vs. data decrypted on a RAW ORAM RO read (left) and Path ORAM access (right) with $Z = 3$. IV_1 and IV_2 are initialization vectors used for encryption.

Path write predictability to ease pipelining. Unfortunately, unless care is taken, the periodic (and expensive) RW accesses cause pipeline stalls requiring that we overprovision encryption units just to handle the RW operation. *Our key observation is: if RW accesses are made to a predictable order of paths, we can pre-compute all encryption operations off the critical path.*

With these techniques combined, RAW ORAM reduces the required encryption bandwidth by $\sim 3\times$. We remark that Ring ORAM [19] also breaks accesses into RO/RW and uses a predictable access pattern for RW accesses. That work, however, does not aim to reduce encryptions on RO accesses and does not recognize or exploit path write predictability.

A. RO and RW Operations

Parameter A. We introduce a new parameter A, set at system boot time. For a given A, RAW ORAM obeys a strict schedule that the ORAM controller performs one RW access after every A RO accesses.

An **RO access** performs *only* the operations needed to read the requested block into the stash, and logically remove it from the ORAM tree. The requested block is then returned or updated as with basic Path ORAM. This corresponds to Steps 2-4 in § II-A with three important changes. First, we will only decrypt the minimum amount of information needed to *find* the requested block and add it to the stash. Precisely, we decrypt the Z block addresses stored in each bucket header (§ II-A), to identify the requested block, and then decrypt the requested block itself (if it is found). The amount of data read vs. decrypted is illustrated in Figure 4. Note that for security, we still read the entire path into the ORAM controller; our optimization is to decrypt less of the path than in Path ORAM.

Second, we add only the requested block to the stash (as opposed to the whole path). Third, we update the bucket header containing the requested block to indicate a block was removed (e.g., by changing its program address to $\perp$), and re-encrypt/write back to memory the corresponding state for

each bucket. To re-encrypt header state only, we encrypt that state with a second initialization vector denoted IV_2. The rest of the bucket is encrypted under IV_1. A strawman design may store both IV_1 and IV_2 in the bucket header (as in Figure 4). We describe an optimized design in § IV-C.

An **RW access** performs a normal (read+writeback) but *dummy* ORAM access to a static sequence of leaves corresponding to a *reverse lexicographic order* of paths (first proposed in [10]). Dummy accesses skip Steps 3-4 in § II-A and read a path randomly chosen without looking up the position map—i.e., their only purpose is to evict blocks from the stash which have accumulated over the A RO accesses.

Security. RO accesses always read paths in the ORAM tree at random, just like Path ORAM. RW accesses occur at predetermined times (always after A RO accesses) and are to predictable/data-independent paths. Thus, RAW ORAM achieves obliviousness assuming the stash does not overflow (discussed in the next section).

B. Performance and Area Characteristics

We compare the memory and encryption bandwidth needed to serve A frontend requests for RAW ORAM and Path ORAM. For both proposals, we assume that bucket headers are stored externally, alongside each bucket and padded to the data block size (64 Bytes). Thus, there are $(L+1)(Z+1)$ blocks on a path. For Path ORAM, both the memory and encryption bandwidth per ORAM access is $2(L+1)(Z_p+1)$ blocks, where Z_p is a competitive bucket size for Path ORAM ($Z_p = 4$ following [16], [23]). We say the relative memory and encryption bandwidth for Path ORAM is $2(Z_p+1)$ blocks.

In RAW ORAM, each RO access reads $(L+1)Z$ on the path, but only decrypts 1 block; it also reads/writes and decrypts/re-encrypts the $L+1$ headers on the path. An RW access reads/writes and decrypts/re-encrypts all the $(L+1)(Z+1)$ blocks on a path. Then, the relative memory bandwidth of RAW ORAM is $Z+2+\frac{2(Z+1)}{A}$, and the relative encryption bandwidth of RAW ORAM is roughly $1+\frac{2(Z+1)}{A}$.

The remaining question for RAW ORAM is: what A and Z combinations result in a stash that will not overflow, yet at the same time minimize encryption and memory bandwidth? We visualize the relative memory band encryption bandwidth of RAW ORAM with different parameter settings in Figure 5. The Ring ORAM paper [19] showed that these parameter settings give negligible stash overflow probability. We see that $Z=5, A=5$ (Z5A5) achieves 6% memory bandwidth improvement and $\sim 3\times$ encryption reduction over Path ORAM. We will use Z5A5 in the evaluation and remark that this configuration requires $C = 64$ to achieve a stash overflow probability of 2^{-80}.

C. Removing AES Bottlenecks With Path Write Predictability

Despite RAW ORAM's theoretic area savings for encryption units, careful engineering is needed to prevent that savings from turning into performance loss. The basic problem is shown in Figure 6 where *w/o AES unit reduction* uses our techniques but is built with the same number of encryption (AES) units as Path ORAM. While there are enough AES units to rate match memory on a RW access, AES units during RO accesses are left idle because RO accesses require less encryptions. In Figure 6 *w AES unit reduction* applies our techniques and uses the desired $3\times$ less AES units. While AES is saturated on RO accesses, DRAM transfers data faster than the available AES bandwidth on RW accesses, which creates a performance bottleneck.

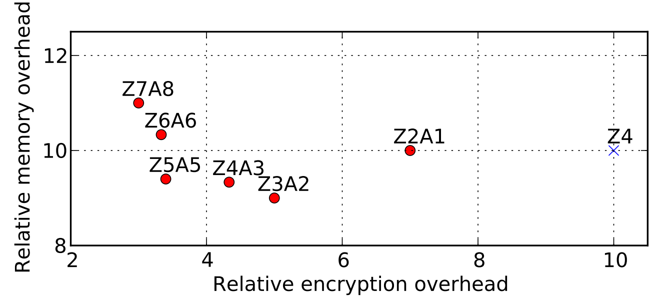

Fig. 5. The relative memory and encryption bandwidth overhead of RAW ORAM with different parameter settings.

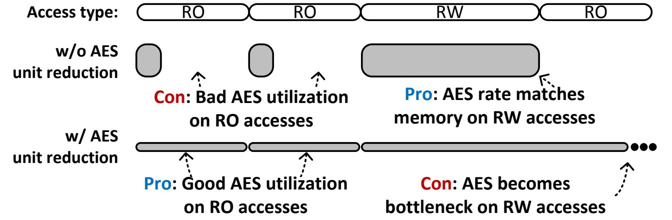

Fig. 6. Potential RAW ORAM performance bottleneck. AES refers to symmetric encryption.

Path Write Predictability. To remove this bottleneck while maintaining the AES unit reduction, we make the following key observation: Since RW accesses occur in a fixed order, we can determine exactly how many times any bucket along any path has been *written* in the past. Suppose G is the number of RW accesses made so far. At startup, the ORAM controller sets $G = 0$ and increments G once after each RW access. The ORAM leaf that is accessed on each RW access is then simply the low-order L bits in G, namely $G \mod 2^L$. (We allocate a 64-bit counter in the ORAM controller to store G.) Now, due to load-balancing nature of reverse lexicographic order, if $\mathcal{P}(l)[i]$ has been written g_i times in the past, then $\mathcal{P}(l)[i+1]$ has been written $g_{i+1} = \lfloor (g_i + 1 - l_i)/2 \rfloor$ where l_i is the i-th bit in leaf l.[2]

Implementation. For the rest of the paper, we assume encryption is implemented using AES-128 in counter/CTR mode. With path write predictability, we know which paths will be read/written on future RW accesses, and can *pre-compute* the AES-CTR initialization vector IV_1 from § IV-A. In other words, we can generate RW AES masks "in the background" during concurrent RO accesses.

To decrypt the i-th 128-bit ciphertext chunk of the bucket with unique ID $BucketID$, as done on an RW ORAM path read, we XOR it with the following mask: $\text{AES}_K(g \mid\mid BucketID \mid\mid i)$ where g is the bucket write count mentioned above. Correspondingly, re-encryption of that

[2]This can be easily computed in hardware as $g_{i+1} = (g_i + {\sim}l_i) \gg 1$, where $\gg$ is a right bit-shift and $\sim$ is bit-wise negation.

chunk on the RW path writeback is done by generating a new mask where the write count has been incremented by 1. We note that with this scheme, g takes the place of IV_1 and since g can be derived internally, we need not store it externally.

On both RO and RW accesses, we must decrypt program addresses and valid bits of all blocks in each bucket (§ IV-A). For this we apply the same type of mask as in Ren et al. [20], namely $\mathsf{AES}_K(\mathrm{IV}_2 \mid\mid BucketID \mid\mid i)$, where IV_2 is stored externally as part of each bucket's header.

At the implementation level, we time-multiplex an AES core between generating masks for IV_1 and IV_2. The AES core prioritizes IV_2 operations; when the core is not servicing IV_2 requests, it generates masks for IV_1 in the background and stores them in a FIFO.

V. Evaluation

We now describe our hardware prototype of Tiny ORAM on a Virtex-7 VC707 FPGA board and analyze its area and performance characteristics.

A. Metrics and Baselines

We evaluate our design in terms of performance and area. Performance is measured as the latency (in FPGA cycles or real time) between when an FPGA user design requests a block and Tiny ORAM *returns* that block. Area is calculated in terms of FPGA lookup-tables (LUT), flip-flops (FF) and block RAM (BRAM), and is measured post place-and-route (i.e., represents final hardware area numbers). For the rest of the paper we count BRAM in terms of 36 Kb BRAM.

We compare Tiny ORAM with two baselines shown in Table II. The first one is Phantom [16]; we will normalize it to our ORAM capacity and the 512 bits/cycle DRAM bandwidth of our VC707 board, and assume no tree top caching. Phantom's performance/area numbers are taken/approximated from the figures in their paper, to our best efforts. The second baseline is a basic Path ORAM with our stash management, to show the area saving of RAW ORAM.

B. Implementation

Parameterization. Both of our designs (Path ORAM and RAW ORAM) use $B = 512$ and $L = 20$. We chose $B = 512$ (64 Bytes) to show that Tiny ORAM can run even very small block sizes without imposing hardware performance bottlenecks. We are constrained to set $L = 20$ because this setting fills the VC707's 1 GB DRAM DIMM, but will discuss working set scaling in §V-E.

Clock regions. The DRAM controller on the VC707 board runs at 200 MHz and transfers 512 bits/cycle. To ensure that DRAM is Tiny ORAM's bottleneck, we optimized our design's timing to run at 200 MHz.

DRAM controller. We interface with DDR3 DRAM through a stock Xilinx on-chip DRAM controller with 512 bits/cycle throughput. From when a read request is presented to the DRAM controller, it takes ~ 30 FPGA cycles to return data for that read (i.e., without ORAM). The DRAM controller pipelines requests. That is, if two reads are issued in consecutive cycles, two 512 bit responses arrive in cycle 30 and 31. To minimize DRAM row buffer misses, we implemented the subtree layout scheme from [20] which allows us to achieve near-optimal DRAM bandwidth (i.e., $> 90\%$, which is similar to Phantom) for our 64 Byte block size.

Encryption. We use "tiny aes," a pipelined AES core that is freely downloadable from Open Cores [1]. Tiny aes has a 21 cycle latency and produces 128 bits of output per cycle. One tiny aes core costs 2865/3585 FPGA LUT/FF and 86 BRAM. To implement the time-multiplexing scheme from § IV-C, we simply add state to track whether tiny aes's output (during each cycle) corresponds to IV_1 or IV_2.

Given our DRAM bandwidth, RAW ORAM requires 1.5 (has to be rounded to 2) tiny aes cores to completely hide mask generation for RW accesses at 200 MHz. To reduce area further, we optimized our design to run tiny aes and associated control logic at 300 MHz. Thus, our final design requires only a single tiny aes core. Basic Path ORAM would require 3 tiny aes cores clocked at 300 MHz, which matches our $3\times$ AES saving in the analysis from § IV-B. We did not optimize the tiny aes clock for basic Path ORAM, and use 4 of them running at 200 MHz.

Data-independent timing and stash scan penalty. Recall from § III-C that we must ensure each ORAM access takes a data-independent amount of time. We will now derive how long Tiny ORAM must stall to respect this requirement. Using the notation from § III-C, tiny aes and our FPGA's DRAM read latency, C' is at least $21 + 30$ cycles (this assumes a 300 MHz AES clock frequency, and does not consider other small latencies introduced by the implementation). Recall from § III-B and § IV-B that, for 2^{-80} stash overflow probability, Path ORAM with $Z = 4$ and RAW Path ORAM with Z5A5 require $C = 78$ and $C = 64$, respectively. Thus, the stash scan algorithm from § III introduces less than $78 + 3 - C' = 30$ cycles and $64 + 3 - C' = 16$ cycles per ORAM access, respectively.

C. Access Latency Comparison

For the rest of the evaluation, all access latencies are averages when running *on a live hardware prototype*. Our RAW ORAM backend can finish an access in 276 cycles ($1.4\mu s$) on average. This is very close to basic Path ORAM; we did not get the 6% theoretical performance improvement because of the slightly more complicated control logic of RAW ORAM.

After normalizing to our DRAM bandwidth and ORAM capacity, Phantom should be able to fetch a 4 KiloByte block in $\sim 60\mu s$. This shows the large speedup potential for small blocks. With bad locality, a 64 Byte block size can improve ORAM latency by $40\times$. We note that Phantom was run at 150 MHz: if optimized to run at 200 MHz like our design, our improvement is $\sim 32\times$. Even with perfect locality where the entire 4 KiloByte data is needed, using a 64 Byte block size introduces only $1.5\times$–$2\times$ slowdown relative to the 4 KiloByte design.

D. Hardware Area Comparison

Our RAW ORAM backend is extremely low area as shown in Table II. The slightly larger control logic in RAW ORAM dampens the area reduction from AES saving. Despite this, RAW ORAM achieves an $\geq 2\times$ reduction in BRAM usage relative to Path ORAM. Note that Phantom [16] did not

TABLE II

PARAMETERS, PERFORMANCE AND AREA SUMMARY OF DIFFERENT DESIGNS. ACCESS LATENCIES FOR PHANTOM ARE NORMALIZED TO 200 MHZ. ALL %S ARE RELATIVE TO THE XILINX XC7VX485T FPGA. FOR PHANTOM AREA ESTIMATES, "∼ 235 + 344" BRAM MEANS 235 BRAM WAS REPORTED IN [16], PLUS 344 FOR TINY AES.

Design	Phantom	Path ORAM	RAW ORAM
Parameters			
Z, A	4, N/A	4, N/A	5, 5
Block size	4 KByte	64 Byte	64 Byte
# of tiny aes cores	4	4	1
Performance (cycles)			
Access 64 B	∼ 12000	270	276
Access 4 KB	∼ 12000	17280	17664
ORAM Backend Area			
LUT (%)	∼ 6000 + 11460	18977 (7%)	14427 (5%)
FF (%)	not reported	16442 (3%)	11359 (2%)
BRAM (%)	∼ 172 + 344	357 (34%)	129 (13%)
Total Area (Backend+Frontend)			
LUT (%)	∼ 10000 + 11460	22775 (8%)	18381 (6%)
FF (%)	not reported	18252 (3%)	13298 (2%)
BRAM (%)	∼ 235 + 344	371 (36%)	146 (14%)

implement encryption: we extrapolate their area by adding 4 tiny aes cores to their design and estimate a BRAM savings of $4\times$ relative to RAW ORAM.

E. Full System Evaluation

For completeness, we evaluate a complete ORAM controller by connecting our RAW ORAM backend to the optimized ORAM frontend proposed in [7]. For our $L = 20$, we add 2 PosMap ORAMs, to attain a small on-chip position map (< 8 KB).

We take memory traces from two real SPEC06-int benchmarks that have very different locality characteristics — libquantum and sjeng — and feed them into the complete frontend + backend design. (Due to optimizations in [7], performance depends on program locality.) libquantum is known to have good locality, and an average our ORAM controller can access 64 Bytes in 490 cycles. sjeng has bad (almost zero) locality and fetching a 64 Byte block requires ~ 950 cycles ($4.75\ \mu s$ at 200 MHz).

VI. CONCLUSION

In this paper we have presented *Tiny ORAM*, a low-latency and low-area hardware ORAM controller. We propose a novel stash management algorithm to unlock low latency and RAW ORAM to decrease area. We demonstrate a working prototype on a Virtex-7 VC707 FPGA board which can return a 64 Byte block in $\sim 1.4\ \mu s$ and requires 5% of the FPGA chip's logic, including the cost of symmetric encryption.

Taken as a whole, our work is an example of how ideas from *circuit design* and *cryptographic protocol design*, coupled with *careful engineering*, can lead to significant efficiencies in practice.

REFERENCES

[1] Open cores. http://opencores.org/.

[2] D. J. Bernstein and T. Lange. The new sha-3 software shootout. Cryptology ePrint Archive, Report 2012/004, 2012. http://eprint.iacr.org/.

[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3, 2009.

[4] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[5] A. Corporation. Protecting the fpga design from common threats. *Whitepaper*.

[6] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.

[7] C. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *Proceedings of the* 20th *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015. Through private correspondance with the authors, they have agreed to release a preprint at: http://csg.csail.mit.edu/pubs/memos/Memo-513/memo513.pdf.

[8] C. Fletcher, M. van Dijk, and S. Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing; an extended version is located at http://csg.csail.mit.edu/pubs/memos/Memo508/memo508.pdf (Master's thesis)*, pages 3–8, Oct. 2012.

[9] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif. Comprehensive evaluation of high-speed and medium-speed implementations of five sha-3 finalists using xilinx and altera fpgas. Cryptology ePrint Archive, Report 2012/368, 2012. http://eprint.iacr.org/.

[10] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing oram and using it efficiently for secure computation. In *Privacy Enhancing Technologies (PET)*, 2013.

[11] O. Goldreich. Towards a theory of software protection and simulation on oblivious rams. In *STOC*, 1987.

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

[13] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[14] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[15] E. Kushilevitz, S. Lu, and R. 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*, pages 143–156. SIAM, 2012.

[16] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. ACM CCS, 2013.

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

[18] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[19] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. V. Dijk, and S. Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. Cryptology ePrint Archive, Report 2014/997, 2014. http://eprint.iacr.org/.

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

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

[22] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[23] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the ACM Computer and Communication Security Conference*, 2013.

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

[25] T. Wollinger, J. Guajardo, and C. Paar. Cryptography on fpgas: State of the art implementations and attacks. *ACM Transactions in Embedded Computing Systems (TECS)*, 2004.

[26] Xilinx. Developing tamper resistant designs with xilinx virtex-6 and 7 series fpgas. *Whitepaper*.

[27] X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the 11th ASPLOS*, 2004.