A Low-Latency, Low-Area Hardware Oblivious RAM Controller

Christopher W. Fletcher†, Ling Ren†, Albert Kwon†,
Marten van Dijk‡, Emil Stefanov°, Dimitrios Serpanos◇, Srinivas Devadas†

† Massachusetts Institute of Technology – {cwfletch, renling, kwonal, devadas}@mit.edu
‡ University of Connecticut – vandijk@engr.uconn.edu
○ University of California, Berkeley – emil@berkeley.edu
◇ Qatar Computing Research Institute, Doha – dserpanos@qf.org.qa

***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 the prior-art implementation. Second, through novel algorithmic and engineering-level optimizations, Tiny ORAM reduces the number of symmetric encryption operations by $\sim 3\times$ compared to a 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 18381 (5%) LUTs and 146 (13%) Block RAM on a Xilinx XC7VX485T FPGA, 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 [13], [15].

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [10], [11], is a cryptographic primitive that completely eliminates the information leakage from a program's 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 [17], [16], [5], [4], [12], [14], [24], [20], [22]. 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, with respect to the size of the untrusted memory, over the chip pins.

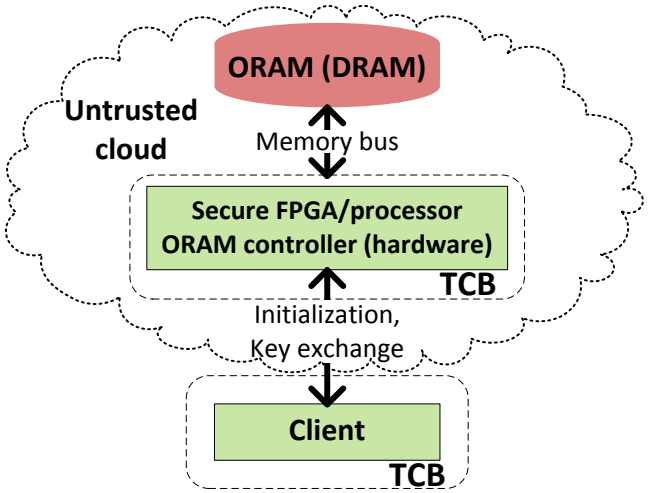

Fig. 1: ORAM in the secure FPGA/processor setting.

An important use case for ORAM is in *trusted hardware* [15], [7], [21], [22], [19], [6]. Figure 1 shows an example target cloud configuration. Here, a client communicates with a trusted secure cloud processor that is attached to untrusted external memory such as disk, DRAM or flash. In 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 [15], [7], [22], [19], [6] have adopted Path ORAM due to its algorithmic simplicity and small client storage. 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 virtually all prior ORAM work, we don't consider leakage over timing, power, RF or other side channels.

A. Opportunities 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], [23]. At the same time, FPGAs have a key performance advantage over software, 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 $5\times$ [8], [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 GByte/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 approximately 250 cycles, compared to over 1000 cycles estimated on ASIC [19], [6].

B. Design Challenges for ORAM on FPGA

The challenge in designing ORAM for FPGA is exactly how to saturate this plentiful memory bandwidth. For instance, the Virtex-7 VC709's 25 GByte/second bandwidth translates to 1024 bits/FPGA cycle (512 bits/cycle per Xilinx MIG memory controller), forcing the ORAM algorithm and implementation to have 1024 bits/FPGA cycle throughput. Recently Maas et al. [15] 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-KByte blocks and will incur huge pipeline stalls with small blocks (§ III). This minimum block size they can support without penalty grows with FPGA memory bandwidth. While benefiting applications with good data locality, a large block size (like 4KBytes in Phantom) severely hurts 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 an FPGA's 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. To illustrate the issue, 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.

[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 AO accesses per EO access

C. 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 main contributions:

Bit-based stash management to enable small block sizes (§ III). We develop a new stash management scheme using efficient bit operations that, when implemented in hardware, removes the block size bottleneck in the Phantom design [15]. 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 access latency by $\geq 40\times$ in the best case compared to Phantom.

RAW ORAM to reduce the required encryption engines (§ IV). We propose a new ORAM scheme called *RAW ORAM* which, through algorithmic and engineering techniques, reduces the number of encryption units by $\sim 3\times$ while maintaining comparable bandwidth to basic Path ORAM.

D. Design and Availability

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 GByte/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 XC7VX485T FPGA's logic and 13% of its on-chip memory. We additionally implemented the ORAM Position Map management techniques due to Fletcher et al. [6] — which increases logic/memory area by only 1%. Thus, the combined design significantly reduces hardware footprint relative to existing alternatives. Our design is open source and available to the community at http://kwonalbert.github.io/oram.

II. Background

As did Phantom, Tiny ORAM originates from and extends Path ORAM [22]. We now explain Path ORAM in detail. Parameters and notations are summarized in Table I.

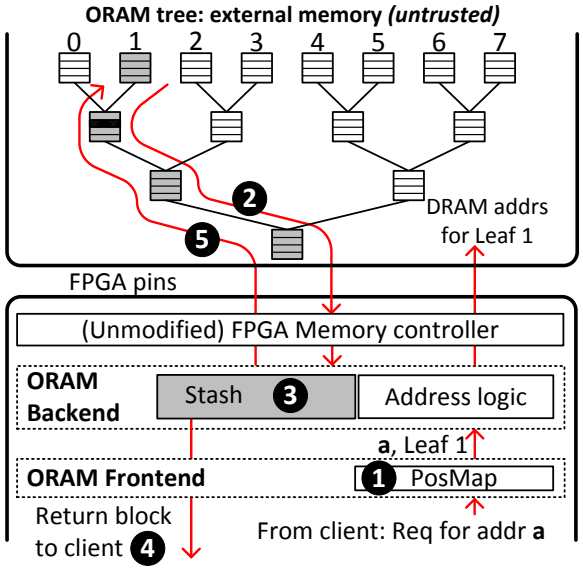

Fig. 2: A Path ORAM of $L = 3$ levels and $Z = 4$ slots per bucket. Suppose block a, shaded black, is mapped to path $l = 1$. At any time, block a can be located in any of the shaded structures (i.e., on path 1 or in the stash).

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 probabilistic encryption (AES-128 counter mode in our design). 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., a BRAM) that stores up to a small number of data blocks. 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.*

Path ORAM access. 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 $\mathsf{accessORAM}(a, op, d')$, where op is either read or write and d' is the new data if op is 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 replace the PosMap entry 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 $op = \mathsf{read}$, return block a to the LLC. If $op = \mathsf{write}$, 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 while keeping the invariant for each block. Fill any remaining space on the path with encrypted dummy blocks.

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 (which is visible to an observer) independent of the program's access pattern. Further, probabilistic encryption ensures that no computationally-bounded adversary can determine which block was requested on each path.

Stash size and eviction. The stash capacity is the maximum number of data blocks on a path plus a small number, i.e., $(L+1)Z + C$. We say that the stash has overflowed if, at the beginning of an ORAM access, the number of blocks in the stash is $> C$. If the stash overflows, some data blocks may be lost, causing ORAM to be functionally incorrect. Therefore, we need to keep stash overflow probability negligible. At the same time, we would like the stash size to be small because on-chip storage is a scarce resource.

Step 5, the eviction step, in the Path ORAM algorithm is essential in keeping the stash size small. This operation tries to move as many blocks as possible from the stash back to the ORAM tree without violating the invariant that each block is somewhere on the path it is mapped to. Intuitively, the more slots there are on each path (i.e., a larger bucket size Z), the more blocks we can put back to the ORAM tree, and the less likely the stash overflows. Yet, a larger bucket size hurts performance (more blocks on a path to read and write). Prior work has shown that the minimum bucket size to ensure negligible stash overflow probability is $Z = 4$ [22], [15]. In this paper, we adopt $Z = 4$ and $C = 78$, which gives 2^{-80} stash overflow probability (using the methodology from [15]).

Not surprisingly, designing efficient logic for the eviction step is non-trivial (more details will be given in § III) and has been a big challenge for hardware Path ORAM designs [15]. One contribution in this paper is a simple and efficient implementation of the eviction logic (see § III).

Bucket header. Implicit in the Path ORAM algorithm, each block is stored alongside some metadata in the ORAM tree. Specifically, each bucket stores a *bucket header*, which tracks the program address and leaf for each of the Z blocks, and an initialization vector (abbreviated IV later on) for probabilistic encryption (e.g., the counter in AES counter mode). Dummy blocks are marked with a special program address $\perp$. Each program address and leaf is roughly L bits and the initialization vector is 64 bits to avoid counter overflow.

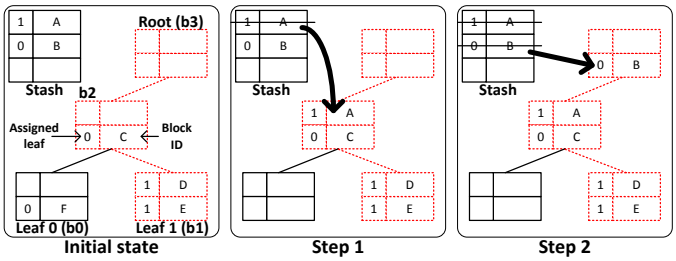

Fig. 3: Stash eviction example for $Z = 2$ slots per bucket. Buckets are labeled $b0, b1, \ldots$ etc. We evict to the path to leaf 1, which includes buckets $b1$, $b2$ and $b3$. Each block is represented as a tuple (leaf, block ID), where leaf indicates which leaf the block is mapped to.

B. Recursive Path ORAM

In basic Path ORAM, the number of entries in the PosMap scales linearly with the number of data blocks in the ORAM. This results in a significant amount of on-chip storage — indeed, Phantom [15] required multiple FPGA's just to store the PosMap for sufficiently large ORAMs. Recursive ORAM was proposed by Shi et al. [20] to solve this problem. The idea is to store the PosMap in a separate ORAM, and store the new ORAM's (smaller) PosMap on-chip. This trick can be recursively applied until the final PosMap is small enough.

Freecursive ORAM [6] demonstrates how to make ORAM recursion basically free with very small area overhead. That paper calls the recursion part Frontend (Step 1 in the above algorithm) and the underlying ORAM Backend (Steps 2-5). Following that terminology, this paper's focus is to optimize the Backend only. Our optimizations can be combined with [6], since techniques of Freecursive ORAM apply to any underlying ORAM Backend. We will evaluate a combined system, using the optimized Frontend of [6], in § V-E for completeness.

III. Stash Management

As mentioned in § II-A, deciding where to evict each block in the stash is a challenge for Path ORAM hardware designs. Conceptually, this operation tries to push each block in the stash as deep (towards the leaves) into the ORAM tree as possible while keeping to the invariant that blocks can only live on the path to their assigned leaf.

Figure 3 works out a concrete example for the eviction logic. In Step 1 of Figure 3, block A is mapped to leaf 1 and therefore may be placed in buckets $b1$, $b2$, and $b3$. It gets placed in bucket $b2$ because bucket $b1$ is full and $b2$ is deeper than $b3$. In Step 2, block B could be placed in $b2$ and $b3$ and gets placed in $b3$ because $b2$ is full (since block A moved there previously).

To decide where to evict blocks, Phantom constructs an FPGA-optimized *heap sort* on the stash [15]. Unfortunately, this approach creates a performance bottleneck because the initial step of sorting the stash takes multiple cycles per block. For example, in the Phantom design, adding a block to the heap takes 11 cycles (see Appendix A of [15]). If the ORAM block size and memory bandwidth is such that writing a block to memory takes less than 11 cycles, system performance is bottlenecked by the heap-sort-based eviction logic and not by memory bandwidth.[2]

In this section, we propose 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 Tricks

Our proposal, the PushToLeaf() routine, is shown in Algorithm 1. PushToLeaf(Stash, l) is run once during each ORAM access and populates an array of pointers $\mathcal{S}$. Stash can be thought of as a single-ported RAM that stores data blocks and their metadata. Once populated, $\mathcal{S}[i]$ points to the block in Stash that will be written back to the i-th position along $\mathcal{P}(l)$. Thus, to complete the ORAM eviction, a hardware state machine sends each block given by Stash$[\mathcal{S}[i]]$ for $i = 0, \ldots, (L+1) * Z - 1$ to be encrypted and written to external memory.

Notations. 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 path l traverses the i-th bucket's left child (0) or right child (1). On Line 3, we initialize each entry of $\mathcal{S}$ to $\perp$, to indicate that the eviction path is initially empty. 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, Occupied)
8:          if a ≠ ⊥ and level > −1 then
9:              offset ← level * Z + Occupied[level]
10:             S[offset] ← i
11:             Occupied[level] ← Occupied[level] + 1
12: function PushBack(l, l', Occupied)
13:     t_1 ← (l ⊕ l')||0                          ▷ Bitwise XOR
14:     t_2 ← t_1 & −t_1                ▷ Bitwise AND, 2C negation
15:     t_3 ← t_2 − 1                              ▷ 2C subtraction
16:     full ← {(Occupied[i] ≟ Z) for i = 0 to L}
17:     t_4 ← t_3 & ∼full                ▷ Bitwise AND/negation
18:     t_5 ← reverse(t_4)                        ▷ Bitwise reverse
19:     t_6 ← t_5 & −t_5
20:     t_7 ← reverse(t_6)
21:     if t_7 ≟ 0 then
22:         return −1                    ▷ Block is stuck in stash
23:     return log_2(t_7)                ▷ Note: t_7 must be one-hot
```

Details for PushBack(). The core operation in our proposal is the PushBack() subroutine, which takes as input the path l we are evicting to, the path l' a block in the stash is mapped to,

[2]Given the 1024 bits/cycle memory bandwidth assumed by the Phantom system, the only way for Phantom to avoid this bottleneck is to set the ORAM block size to be $\geq$ 1408 Bytes.

and outputs which level on path l that block should get written back to. In Line 13, t_1 represents in which levels the paths $P(l)$ and $P(l')$ diverge. In Line 14, t_2 is a one-hot bus where the 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 (due to previous calls to PushBack()). Finally, Lines 18-20 turn all current bits off except for the *left-most set bit*, which now indicates the level furthest towards the leaves that the block can be pushed back to.

Security. We remark that while our stash eviction procedure is highly-optimized for hardware implementation, it is algorithmically equivalent to the original stash eviction procedure described in Path ORAM [22]. Thus, security follows from the original Path ORAM analysis.

B. Hardware Implementation and Pipelining

Algorithm 1 runs $C + (L+1)Z$ iterations of PushBack() per ORAM access. In hardware, we pipeline Algorithm 1 in three respects to hide its latency:

First, the PushBack() circuit itself is pipelined to have 1 block / cycle throughput. PushBack() itself synthesizes to simple combinational logic where the most expensive operation is two's complement arithmetic of $(L+1)$-bit words (which is still relatively cheap due to optimized FPGA carry chains). reverse() costs no additional logic in hardware. 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. An important subtlety is that we don't add pipeline stages between when Occupied is read and updated. Thus, a new iteration of PushBack() can be started every cycle.

Second, as soon as the leaf for the ORAM access is determined (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 external memory.* Since a new block can be processed by PushBack() each cycle, eviction logic will not be the system bottleneck.

IV. Minimizing Design Area

Another serious problem for ORAM design is the area needed for encryption units. Recall from § II-A that all data touched 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 [15] 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 address this problem we now propose a new ORAM design, which we call *RAW ORAM*, optimized to minimize encryption bandwidth at the algorithmic and engineering level.

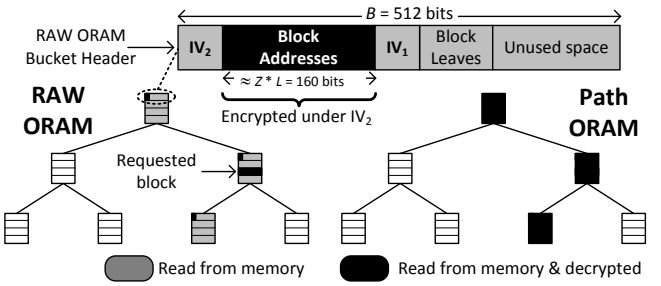

Fig. 4: Data read vs. data decrypted on a RAW ORAM AO read (left) and Path ORAM access (right) with $Z = 3$. IV_1 and IV_2 are initialization vectors used for encryption.

A. RAW ORAM Algorithm

First, we split ORAM Backend operations into two flavors: *access-only* (AO) and *eviction-only* (EO) accesses. AO accesses perform the minimal amount of work needed to service a client processor's read/write requests (i.e., last level cache misses/writebacks) and EO accesses perform evictions (to empty the stash) in the background. To reduce the number of encryption units needed by ORAM, we optimize AO accesses *to only decrypt the minimal amount of data needed to retrieve the block of interest, as opposed to the entire path.* EO accesses require more encryption/decryption, but occur less frequently.[3] We now describe the protocol in detail:

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 EO access after every A AO accesses.

An **AO access** reads an ORAM tree path (as does Path ORAM) but only performs the minimal number of on-chip operations (e.g., decryption) needed to decrypt/move the requested block into the stash and logically remove that block from the ORAM tree. 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.

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-D.

[3]We remark that Ring ORAM [18] also breaks ORAM into AO/EO and uses a predictable access pattern for EO accesses. That work, however, does not aim to reduce encryptions on AO accesses and does not recognize or exploit path eviction predictability.

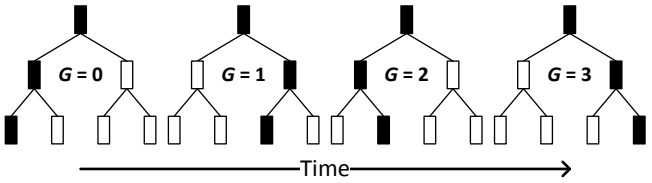

Fig. 5: Reverse lexicographic order of paths used by EO accesses. After the $G = 3$ path is evicted to, the order repeats.

An **EO access** performs a normal but *dummy* Path ORAM access to a static sequence of leaves corresponding to a *reverse lexicographic order* of paths (first proposed in [9]). By dummy access, we simply mean reading and evicting a path to push out blocks in the stash that have accumulated over the A AO accesses (as described in [19]).

We give a visual example for the reverse lexicographic order in Figure 5. Here, G is a counter that tracks the number of EO accesses made so far. Notice that the leaf (and its corresponding path) being evicted to next is given precisely by $G \mod 2^L$ if we encode leaves as described in § III-A.

Security. We note that AO accesses always read paths in the ORAM tree at random, just like Path ORAM. Further, EO accesses occur at predetermined times (always after A AO accesses) and are to predictable/data-independent paths.

B. Performance and Area Characteristics

Assume for simplicity that the bucket header is the same size as a data block (which matches our evaluation). Then, each AO access reads $(L+1)Z$ blocks on the path, but only decrypts 1 block; it also reads/writes and decrypts/re-encrypts the $L+1$ headers/blocks. An EO access reads/writes and decrypts/re-encrypts all the $(L+1)(Z+1)$ blocks on a path. Thus, in RAW ORAM the relative memory bandwidth per bucket is $Z+2+\frac{2(Z+1)}{A}$, and the relative encryption bandwidth per bucket 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? In Figure 6, we visualize the relative memory and encryption bandwidth of RAW ORAM with different parameter settings that have been shown [18] to give negligible stash overflow probability. We find $Z = 5, A = 5$ (Z5A5) to be a good trade-off as it 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 $C = 64$ for a 2^{-80} stash overflow probability.

C. Design Challenges In Practice

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 that by reducing encryption units (we say AES from now on) to provide "just enough" bandwidth for AO accesses, we are forced to wait during EO accesses for that reduced number of AES units to finish decrypting/re-encrypting the entire path. Further, since all AES IVs are stored externally with each bucket, the AES units can't start working on a new EO access until that access starts.

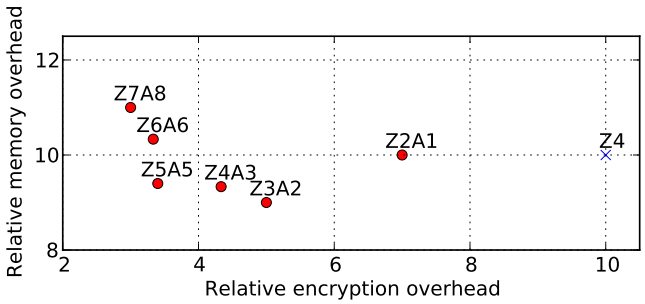

Fig. 6: The relative memory and encryption bandwidth overhead of RAW ORAM with different parameter settings.

D. Path Eviction Predictability

To remove the above bottleneck while maintaining the AES unit reduction, we make the following key observation: *Since EO accesses occur in a predictable, fixed order, we can determine exactly how many times any bucket along any path has been* written *in the past.*

Recall that G indicates the number of EO accesses made so far. (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 evicted to g_i times in the past, then $\mathcal{P}(l)[i+1]$ has been evicted $g_{i+1} = \lfloor (g_i + 1 - l_i)/2 \rfloor$ where l_i is the i-th bit in leaf l. 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.

Using eviction predictability, we will *pre-compute* the AES-CTR initialization vector IV_1 (§ IV-A). Simply put, this means the AES units can do all decryption/encryption work for EO accesses "in the background" during concurrent AO accesses.

To decrypt the i-th 128-bit ciphertext chunk of the bucket with unique ID $BucketID$ at level j in the tree, as done on an EO ORAM path read, we XOR it with the following mask: $\mathsf{AES}_K(g_j \mid\mid BucketID \mid\mid i)$ where g_j is the bucket eviction count defined above. Correspondingly, re-encryption of that chunk is done by generating a new mask where the write count has been incremented by 1. We note that with this scheme, g_j takes the place of IV_1 and since g_j can be derived internally, we need not store it externally.

On both AO and EO accesses, we must decrypt the 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. [19], 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 perfor-

mance characteristics. Our complete design, built on top of the Recursive ORAM design due to [6], is open source and available to the community at http://kwonalbert.github.io/oram.

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. For all experiments, we compiled with Xilinx Vivado version 2013.4.

We compare Tiny ORAM with two baselines shown in Table II. The first one is Phantom [15], which we normalize to our ORAM capacity and the 512 bits/cycle DRAM bandwidth of our VC707 board. We further disable Phantom's 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 technique, to show the area saving of RAW ORAM.

B. Implementation

Parameterization. Both of our designs (Path ORAM and RAW ORAM) use $B = 512$ bits per block and $L = 20$ levels. 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 GByte DRAM DIMM.

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 [19] 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-D, 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 EO 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.

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 KByte block in $\sim 60\mu s$. This shows the large speedup potential for small blocks. Suppose the program running has bad data locality (i.e., even though Phantom fetches 4 KBytes, only 64 Bytes are touched by the program). In this case, Tiny ORAM using a 64 Byte block size improves ORAM latency by $40\times$ relative to Phantom with a 4 KByte block size. 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 KByte data is needed, using a 64 Byte block size introduces only $1.5 - 2\times$ slowdown relative to the 4 KByte design.[4]

D. Hardware Area Comparison

In Table II, we also see that the RAW ORAM Backend requires only a small percentage of the FPGA's total area. 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 [15] did not 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

We now evaluate a complete ORAM controller by connecting our RAW ORAM Backend to the optimized ORAM Frontend (called 'Freecursive ORAM') proposed in [6]. For completeness, we also implemented and evaluated a baseline Recursive Path ORAM. (To our knowledge, we are the first to implement any form of Recursive ORAM in hardware.) For our $L = 20$, we add 2 PosMap ORAMs, to attain a small on-chip position map (< 8 KB).

Figure 7 shows the average memory access latency of several real SPEC06-int benchmarks. Due to optimizations in [6], performance depends on program locality. For this reason, we also evaluate two synthetic traces: scan which has perfect locality and rand which has no locality. We point out two extreme benchmarks: libq is known to have good locality, and on average our ORAM controller can access 64 Bytes in 490

[4]We remark that this slowdown is due to the Path ORAM algorithm: with a fixed memory size, a larger block size results in a shorter ORAM tree (i.e., L decreases which improves performance).

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, "$\sim$ 235 + 344" BRAM means 235 BRAM was reported in [15], 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	$\sim$ 12000	270	276
Access 4 KB	$\sim$ 12000	17280	17664
ORAM Backend Area			
LUT (%)	$\sim$ 6000 + 11460	18977 (7%)	14427 (5%)
FF (%)	not reported	16442 (3%)	11359 (2%)
BRAM (%)	$\sim$ 172 + 344	357 (34%)	129 (13%)
Total Area (Backend+Frontend)			
LUT (%)	$\sim$ 10000 + 11460	22775 (8%)	18381 (6%)
FF (%)	not reported	18252 (3%)	13298 (2%)
BRAM (%)	$\sim$ 235 + 344	371 (36%)	146 (14%)

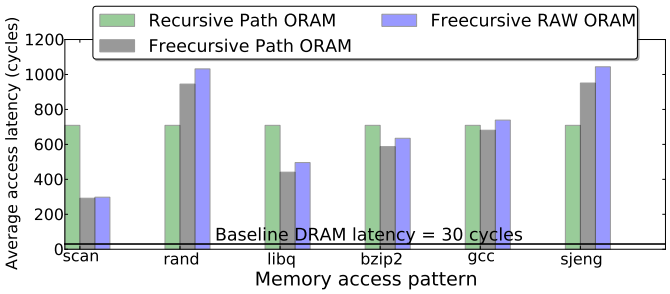

Fig. 7: Average number of FPGA clock cycles needed to complete an ORAM access.

cycles. sjeng has bad (almost zero) locality and fetching a 64 Byte block requires $\sim$ 950 cycles (4.75 μs at 200 MHz). Benchmarks like sjeng reinforce the need for small blocks: setting a larger ORAM block size will strictly decrease system performance since the additional data in larger blocks won't be used.

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 design area. We demonstrate a working prototype on a Virtex-7 VC707 FPGA board which can return a 64 Byte block in $\sim$ 1.4 μs 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.

Acknowledgments: We thank the anonymous reviewers for many constructive comments. This research was partially supported by QCRI under the QCRI-CSAIL partnership and by the National Science Foundation. Christopher Fletcher was supported by a DoD National Defense Science and Engineering Graduate Fellowship.

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] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.
[6] 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 *ASPLOS*, 2015.
[7] C. Fletcher, M. van Dijk, and S. Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *STC*, 2012.
[8] 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/.
[9] 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 *PET*, 2013.
[10] O. Goldreich. Towards a theory of software protection and simulation on oblivious rams. In *STOC*, 1987.
[11] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. In *Journal of the ACM*, 1996.
[12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.
[13] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.
[14] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *SODA*, 2012.
[15] 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. In *CCS*, 2013.
[16] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.
[17] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, 1997.
[18] 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/.
[19] 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 *ISCA*, 2013.
[20] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt*, 2011.
[21] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *S&P*, 2013.
[22] 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 *CCS*, 2013.
[23] Altera Corporation. Protecting the fpga design from common threats. *Whitepaper*.
[24] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.
[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 *ASPLOS*, 2004.