Design and Implementation of the Ascend Secure Processor

As Published: 10.1109/TDSC.2017.2687463

Publisher: Institute of Electrical and Electronics Engineers (IEEE)

Persistent URL: https://hdl.handle.net/1721.1/135751

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Design and Implementation of the Ascend Secure Processor

Ling Ren Christopher W. Fletcher Albert Kwon Marten van Dijk Srinivas Devadas

***Abstract*—This paper presents post-silicon results for the Ascend secure processor, taped out in a 32 nm SOI process. Ascend prevents information leakage over a processor's digital I/O pins — in particular, the processor's requests to external memory — and certifies the program's execution by verifying the integrity of the external memory. In secure processor design, encrypting main memory is not sufficient for security because *where* and *when* memory is accessed reveals secret information. To this end, Ascend is equipped with a hardware Oblivious RAM (ORAM) controller, which obfuscates the address bus by reshuffling memory as it is accessed. To our knowledge, Ascend is the first prototyping of ORAM in custom silicon. Ascend has also been carefully engineered to ensure its timing behaviors are independent of user private data. In 32 nm silicon, all security components combined (the ORAM controller, which includes 12 AES rounds and one SHA-3 hash unit) impose a moderate area overhead of 0.51 mm^2. Post tape-out, the security components of the Ascend chip have been successfully tested at 857 MHz and 1.1 V, at which point they consume 299 mW of power.**

I. Introduction

As embedded and mobile devices become ubiquitous, users have become more computationally limited and computation outsourcing is becoming more common. From financial information to medical records, sensitive user data is being sent to and computed upon by the cloud. Data privacy has therefore become a huge concern, as sensitive user data is being revealed to and can be attacked by potentially malicious cloud applications, hypervisors/operating systems, or insiders.

One example is health diagnoses given a user's private medical records [31]. In this case, a mobile device is constantly monitoring its user's symptoms and wants to compute the likelihood that the user has some condition given these symptoms. To outsource computation, the device sends a cloud server an encryption of {symptoms, condition of interest}, which will be inputs to a program (call it `MedComp()`). After running `MedComp()` on the user's private data, the server sends an encryption of the result (e.g., "there is a 55% likelihood that you have the condition") back to the user. To maintain privacy, the server must never learn anything about the user's private inputs — the symptoms or diseases of interest — at any time before, during or after the computation. More examples are given in [31].

One candidate solution is for the server to use tamper-resistant hardware [20], [26], [41]. Here, computation takes place inside a secure hardware compartment on the server side that protects the user's private data while it is being computed upon. The smaller the trusted computing base (TCB), the better from a security perspective. At the same time, removing components from the TCB typically impacts performance and/or energy efficiency. Despite the hit in efficiency, the computationally-limited user is still motivated to outsource computation since compute, energy and memory resources are significantly cheaper for the server than the user.

A serious limitation with current secure hardware solutions, however, is that they have to "trust" the program running on the secure hardware. To "trust" the program, the user has to "sign-off" on that program, believing that it is free of malicious code or bugs that leak sensitive data. But applications like `MedComp()` are seldom trustworthy. Verifying bug-free and malicious behavior is a hard problem, if not intractable, for sufficiently complex programs. Frequent software patches, which are typical in modern software development, only confound the issue. Furthermore, the user may not have access to the program in the first place as the program may be proprietary.

The Ascend[1] secure processor is a step towards solving this problem. With Ascend, the user sends the server encrypted input (symptoms and medical information), and requests a program (e.g., `MedComp()`) to be executed. The server would then run the program on the user's private data on an Ascend processor, which decrypts user input, runs the program and returns encrypted results to the user. Ascend ensures that even a buggy or malicious program cannot leak information about the user's private data outside the Ascend chip.

A. The Problem with Untrusted Programs

Today's secure processors (e.g., Intel+TXT [20], XOM [26], Aegis [41], and Intel SGX [7]) leak private information when running badly written or malicious programs. Consider the following scenario. A user sends the server its symptoms and condition of interest (denoted `x`) and expects the server to run the `MedComp()` program described above. Note, the user will send `x` to the server encrypted, and `x` is decrypted inside the secure processor. However, the server being curious may instead run the following program in order to learn the user's data.

```
void curious(x) {
  if (x & 0x1) M[0];
}
```

`M` denotes program memory, which will be backed up in cache (inside the processor) and in DRAM/disk (outside the processor). If `M[0]` is not in cache, whether it is accessed is visible to the server and reveals a bit in the user's data. The server can then repeat this experiment with different versions of `curious(x)` to learn other bits in `x`.

These types of attacks are difficult to prevent. Adding cache doesn't help: the server can re-design `curious(x)` to miss the cache. Encrypting data that leaves the secure processor

[1] Ascend stands for Architecture for Secure Computation on ENcrypted Data.

also doesn't help. As shown in the above example, the attack can succeed when the server sees *where* the program accesses memory (the access pattern), and whether/when accesses occur.

Of course, even if the server does run a non-malicious program, program characteristics or bugs can leak private information in the same way. Here, we use a toy example `curious()` to illustrate the point, but several works have demonstrated information leakage from memory access patterns in practical scenarios [48], [23], [45], [21].

B. Ascend: Obfuscation in Hardware

Ascend defeats attacks like those posed by the `curious()` program by performing obfuscation in hardware. Ascend guarantees that given an untrusted program P, a public length of time T and two arbitrary inputs x and x': running $P(x)$ for T time is indistinguishable from running $P(x')$ for T time from any of the Ascend processor's digital external pins. Ascend has input/output (I/O) and address pins like a normal processor, but obfuscates both the value and the timing of digital signals on these pins. Therefore, the server does not gain any information by watching these pins. Ascend does not protect analog side channels like power, heat, electromagnetic or radio-frequency channels.

Unlike a normal processor, Ascend runs for a parameterizable amount of time T that is chosen by the user before the program starts, and is public to the server. Since T is set a priori, it may or may not be sufficient to complete the program given the user's inputs. If the program terminates before T time has elapsed, Ascend will perform indistinguishable dummy work until time T. After T time is complete, Ascend will output either the final result or an intermediate program state, encrypted with the user's key. In either case, the Ascend chip emits the same number of encrypted bits (the result or state, possibly padded). The server, therefore, cannot determine whether the program completes or how much forward progress was made. In either case, signals on Ascend's digital pins must leak no information about private user data for the entire T time, regardless of the program being run.

Putting these ideas together, the server cannot learn about the user's input by running programs like `curious()` from Section I-A. Regardless of whether and how `curious()` accesses external memory, Ascend runs for T time and behaves in an indistinguishable way from the perspective of digital side channels.

C. Contributions

This paper presents post-silicon results for the Ascend secure processor. We make three primary contributions:

1) We give an overview of the Ascend execution model to securely run *untrusted* programs. This part (Section II) has been published in an STC workshop paper [14].
2) We provide a comprehensive overview of challenges in implementing a hardware Oblivious RAM (ORAM) controller, the core component in the Ascend design. We present new techniques to address these issues. These materials (Section IV, V, VI) have been published in a series of our papers [35], [11], [12].
3) We implement all the techniques described in this paper in hardware, tape out the Ascend processor in 32 nm silicon, and report hardware evaluation and measurement results. This part (Section VII) is new material for this manuscript.

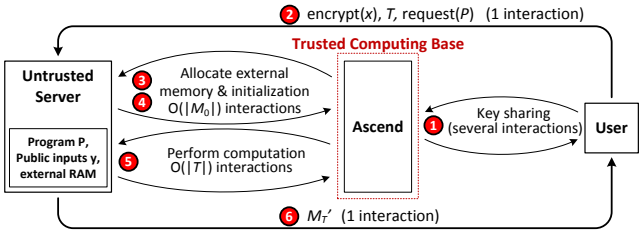


Fig. 1. The protocol between a user, an untrusted server and Ascend. Numbers follow Section II-A.

The security mechanisms of Ascend take up 0.51 mm^2 of silicon in 32 nm technology, which is roughly half the area of a single processor core. Post tape-out, the security components of the Ascend chip has been successfully tested at 857 MHz and 1.1 V, at which point they consume 299 mW of power. Our design is entirely written in Verilog and is available at https://github.com/ascend-secure-processor/oram.

D. Organization

The rest of the paper is organized as follows. Section II presents the Ascend execution model. Section III gives a background on Oblivious RAM (ORAM). Section IV presents the main challenges in implementing ORAM in hardware. Sections V and VI describe optimizations which address the aforementioned challenges. Sections VII present ASIC implementation and measurement results. Section VIII discusses related work. Finally, Section IX concludes.

II. Framework

A. Protocols

The user wants to perform computation on private inputs x using a program P. P is stored on the server-side and may have a large amount of public data y associated with it (e.g., the contents of a database). The result of the user's computation is denoted $P(x, y)$. P is assumed to be a batch program, meaning that it only takes input at the beginning, and produces an encrypted output upon termination.[2] The protocol for computing $P(x, y)$ works as follows (shown graphically in Figure 1):

1) The user shares a symmetric key K securely with Ascend, using standard secure channel protocols (e.g., as used by Intel's SGX [7]). For this purpose, we assume that Ascend is equipped with a private key and a certified public key.

[2]For interested readers, the model is generalized to programs that take public streams of input [46], such as content-based image recognition software processing video feeds.

2) The user encrypts its inputs x using K, and then chooses T and R. T is the *public* time budget that the user is willing to pay the server to compute on P. R is the number of bytes reserved for P's final or intermediate output. The user sends $(\mathsf{encrypt}_K(x), T, R, \mathsf{request}(P, y))$ to the server, where $\mathsf{request}(P, y)$ is a request to use the program P with server data y.
3) After receiving $(\mathsf{encrypt}_K(x), T, R, \mathsf{request}(P, y))$, the server sends $\mathsf{encrypt}_K(x), P, y, T, R$ to Ascend. Ascend decrypts $\mathsf{encrypt}_K(x)$, and writes x, P and y to external memory, (re-)encrypted.
4) Ascend spends T cycles running P. After T cycles, Ascend obtains r, the R-byte result that equals either $P(x, y)$ or some intermediate result.
5) Ascend creates a Message Authentication Codes (MAC, e.g. a keyed hash [4]) on the program, user's input and parameters to certify the execution, i.e., $h = \mathsf{MAC}(P \parallel x \parallel y \parallel T)$.[3]
6) Ascend sends $\mathsf{encrypt}_K(r)$ and h back to the user (via the server).
7) The user verifies the MAC, (if match) decrypts and checks whether the program finished. Without loss of generality, we assume r contains an "I am done" message if P finishes.

B. Threat Model

Ascend is a single-chip coprocessor on the server and interacts with the server to run programs. The session key K is stored in a register, accessible only to the encrypt/decrypt units — not to the program P. The Ascend chip is assumed to be tamper-resistant: the server cannot remove packaging/metal layers, and hence cannot see K or any decrypted values inside Ascend. The server can, however, monitor the traffic and timing on Ascend's I/O pins. The I/O pins record Ascend's interactions to an external memory while the program is running — including the data, operation and address of those requests as well as when those requests are made — and also record when the program terminates and the final program output. In this paper, we assume all external memory requests are to fetch and evict processor cache lines from/to main memory. Ascend does not protect any analog side channels (e.g., power, heat, electromagnetic or radio-frequency).

The server can repeatedly perform experiments with Ascend. For each experiment, the server initializes Ascend with arbitrary $\mathsf{encrypt}_K(x)$, P, y, T, R and monitors how Ascend interacts with the outside world through the digital pins. We make no assumptions as to how the server monitors the pins. An insider may attach an oscilloscope to the pins directly, or create a sniffer program that monitors what bits change in Ascend's external memory. At any time, the server can modify the contents in the external memory or perform a denial-of-service attack by delaying or not returning memory responses. These experiments can be run offline without the user's knowledge. Running the `curious()` program from Section I-A is one such experiment.

To cheat the user, the server can initialize the system incorrectly (e.g., supply a different program P', run for less than T time), or tamper with external memory during execution. The certified execution along with the memory integrity verification in Section V-B will allow the user to detect any cheating. Given that the user can detect cheating, the server's motivation changes: it wants as much business from the user as possible and therefore is motivated to return correct results. But it still wishes to *passively* (without actively tampering) learn as much information as possible about the user's input.

C. Design Overview

Ascend communicates over its I/O pins to a fixed-size external memory that is controlled by and visible to the server. All data sent to external memory from Ascend is encrypted under K. Addresses sent to external memory, and whether a given operation is a read or a write (the *opcode*), are in plaintext. Preventing information leakage from the plaintext address and opcode is a major goal of Ascend. Address and opcode obfuscation is accomplished using a hardware Oblivious RAM (ORAM) controller. At a high-level, the ORAM controller encrypts and shuffles memory such that any two access patterns of the same length are indistinguishable to the server.

To obfuscate the timing channel, we architect Ascend to access external memory at fixed/data-independent rate. The data-independent rate may cause the external memory to be accessed when not needed. In this case, Ascend performs a dummy access, which is indistinguishable from a real access due to the guarantees provided by ORAM. On the other hand, the program may wish to access memory sooner than the data-independent rate allows. In this case, the program will stall until the next access is allowed to happen. This can be achieved with a simple hardware counter and queue that increments each processor cycle and triggers a real or dummy memory access when it reaches the threshold. The threshold can be set/changed by the server before or during the computation. Padding program execution time to the threshold T is handled similarly: dummy ORAM accesses are made, at the prescribed interval, until T is reached. We also engineer the ORAM controller microarchitecture carefully such that *during each ORAM access*, the timing of each request made to the DRAM does not depend on any user secret.

III. Path ORAM

Our hardware ORAM is based on the Path ORAM algorithm due to its simplicity and efficiency [40]. We will first define ORAM [19] and then describe the Path ORAM algorithm in detail.

A. ORAM Security Definition

An ORAM is made up of two components: a (trusted) client and (untrusted) external memory or server. In the Ascend setting, the client is an ORAM controller — trusted logic

[3] If P is the server's proprietary software, the server can have a trusted third party certify $\mathsf{hash}(P)$, and replace P with $\mathsf{hash}(P)$ in the MAC. This protects the server from revealing the detailed code of P. Same applies to the dataset y.

TABLE I
PATH ORAM PARAMETERS AND NOTATIONS.

Notation	Meaning
N	Number of real data blocks in ORAM
L	Depth of the ORAM tree
Z	Maximum number of real blocks per bucket
B	Data block size (in bits)
A	Eviction rate (larger means less frequent)
G	Eviction counter
H	The number of ORAMs in the recursion
X	The number of leaves stored per PosMap block
C	Maximum stash occupancy (not counting transient path)
$\mathcal{P}(l)$	Path from root to leaf l
K	Symmetric key established at program start

on the processor die which intercepts last-level cache (LLC) misses. Server storage is a DRAM DIMM which the server uses as main memory.

ORAM guarantees the following. Suppose the processor creates two sequences of memory requests (LLC misses) W and W'. Each sequence is made up of read $(\mathsf{read}, \mathsf{addr})$ and write $(\mathsf{write}, \mathsf{addr}, \mathsf{data})$ tuples. The ORAM controller guarantees that from the server's perspective (which can monitor the processor bus, or the state of the DRAM): if $|W| = |W'|$, where $|\ldots|$ indicates length, then W is computationally indistinguishable from W'. Informally, this hides the tuples in W and W': namely whether the client is reading/writing to the storage, where the client is accessing, and the underlying data that the client is accessing.

B. Basic Path ORAM Protocol

Path ORAM's [40] server storage is logically structured as a binary tree [38], as shown in Figure 2. The ORAM tree's levels range from 0 (the root) to L (the leaves). Each node in the tree is called a *bucket* and has a fixed number of slots (denoted Z) which can store B-bit data blocks. A slot may be empty at any point, in which case we say the slot contains a *dummy block*. Non-empty slots contain *real blocks*. All blocks in the tree (real or dummy) are encrypted with a probabilistic encryption scheme, so any two blocks are indistinguishable after encryption. A *path* through the tree is a contiguous sequence of buckets from the root to some leaf l and is referred to as $\mathcal{P}(l)$. A path $\mathcal{P}(l)$ is uniquely identified by a leaf l, so we use "path" and "leaf" interchangeably in the paper. All symbols and parameters related to Path ORAM are summarized in Table I for convenience.

The client logic is made up of the *position map*, the *stash* and control logic. The position map, PosMap for short, is a lookup table that associates each data block with a random path in the ORAM tree. If N is the maximum number of real data blocks in the ORAM, the PosMap capacity is $N \cdot L$ bits: one mapping per block. The stash is a memory that temporarily stores up to a small number of data blocks (in plaintext).

1) Path ORAM invariant and operation: At any time, each data block in Path ORAM is mapped to a random path via the PosMap. Path ORAM maintains the following invariant:

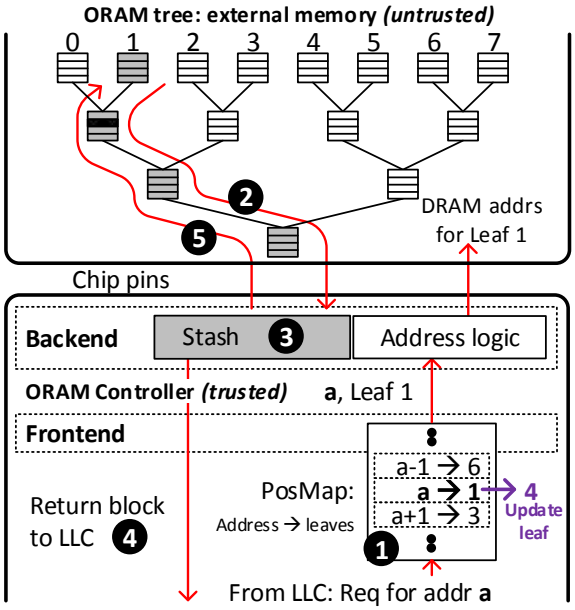


Fig. 2. A Path ORAM of $L = 3$ levels and $Z = 4$ slots per bucket. Suppose block a, shaded black, is mapped to $\mathcal{P}(1)$. Block a can be located in any of the shaded structures (i.e., on path 1 or in the stash).

If a block is mapped to leaf l, then it must be either in some bucket on $\mathcal{P}(l)$ or in the stash.

To make a request for a block with address a (block a for short), the client calls the function $\mathsf{accessORAM}(a, op, d')$, where op is either read or write and d' is the new data when $op = \mathsf{write}$ (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 label l' and update 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 and discard the dummies. 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 client. 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 $\mathcal{P}(l)$ 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.

Some metadata is stored alongside each block in the ORAM tree and in the stash. Metadata in the ORAM tree are encrypted while metadata in the stash are in plaintext. The metadata includes a block's address and its leaf label. A special address $\perp$ is reserved for dummy blocks. These metadata allow the ORAM controller to discard dummy blocks, find the requested block, and evict blocks in the above steps.

The eviction step (Step 5) warrants more detailed explanation. Conceptually, this step 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 a block can only live on its assigned path. Figure 3 works out a concrete example for the eviction logic. In Step 1 of Figure 3, block A

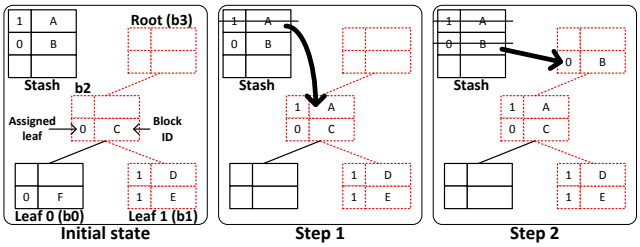


Fig. 3. Stash eviction example for $Z = 2$ slots per bucket. Buckets are labeled $b0, b1, \ldots,$ etc. We evict along the path to leaf 1, which includes buckets $b1$, $b2$ and $b3$. Each block is represented as a tuple (path, block ID), where 'path' indicates which path the block is mapped to.

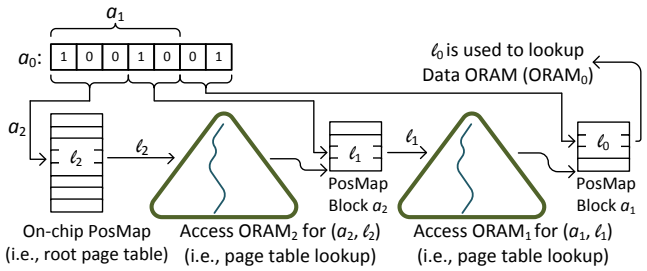


Fig. 4. Recursive ORAM with PosMap block sizes $X = 4$, making an access to the data block with program address $a_0 = 1001001_2$. Recursion shrinks the PosMap capacity from $N = 128$ to 8 entries.

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).

We refer to Step 1 (the PosMap lookup) as the $\mathsf{Frontend}(a)$, or Frontend, and Steps 2-5 as the $\mathsf{Backend}(a, l, l', op, d')$, or Backend. We will describe in detail optimizations we make to these modules in Section V and VI, respectively.

2) Bucket format: Each bucket is $Z(L + O(L) + B)$ bits in size before encryption. Here, L bits denote the path each block is assigned to, $O(L)$ denotes the (encrypted) logical address a for each block. In practice, $O(L) = L + 1$ or $L + 2$ bits for $Z = 4$, which means 50% of the DRAM can be used to store real blocks [35]. Conceptually, we can reserve a unique logical address $\perp$ to mark a bucket slot as containing a dummy block. Our actual implementation use Z extra 'valid' bits per bucket.

We use AES-128 in counter mode for encryption. The ORAM controller maintains a monotonically increasing (global) counter IV in a dedicated register.[4] To encrypt a bucket:

1) Break up the plaintext bucket into 128-bit chunks. Encrypt each chunk with the following one-time pad: $\mathsf{AES}_K(\mathsf{IV} \parallel i) \oplus chunk_i$, where $\parallel$ denotes concatenation.
2) The current IV is written out alongside the encrypted bucket.
3) $\mathsf{IV} \leftarrow \mathsf{IV} + 1$.

IV may be initialized to 1 during ORAM initialization. Thus, it is important to use a different session key K for each run, to avoid a replay attack. After encryption, each bucket is $Z(L + O(L) + B) + |\mathsf{IV}|$ bits in size.

3) ORAM initialization: One may initialize ORAM simply by zeroing-out main memory. This means all IV fields are also 0: AES units performing bucket decryption should treat the bucket as fully empty when IV equals 0. Our actual implementation uses this method. The downside of this scheme is that it requires $O(N)$ work upfront, for every program execution. One could also perform a "lazy initialization" scheme, which gradually initializes each bit of memory as it is accessed the first time [35].

[4]We could use per-bucket counters (as proposed in [35], but that would introduce a security flaw when combined with our PMMAC technique in Section V-B, cf. [11].

4) Security: The intuition for Path ORAM's security is that every PosMap lookup (Step 1) will yield a fresh random leaf to access the ORAM tree for that access. This makes the sequence of ORAM tree paths accessed independent of the actual program address trace. Probabilistic encryption hides *which* block is accessed on the path. Further, stash overflow probability is negligible if $Z \geq 4$ [40], [30]. We assume $Z = 4$ for the rest of the paper.

We remark that ORAM does not hide the total number of memory accesses. This is not an issue in our setting because the total number of accesses is fully determined by the runtime budget T and the memory access interval, independent of any private user data (Section II-C).

C. Recursive ORAM

As mentioned in previous sections, the number of entries in the PosMap scales linearly with the number of data blocks in the ORAM. In the secure processor setting, this results in a significant amount of on-chip storage (up to hundreds of MegaBytes). To address this issue, Shi et al. [38] proposed a scheme called *Recursive ORAM*. The basic idea is to store the PosMap in a separate ORAM, and store the new ORAM's (smaller) PosMap on-chip. If the new on-chip PosMap is still too large, additional ORAMs can be added. *We make an important observation that the mechanics of Recursive ORAM are remarkably similar to multi-level page tables in traditional virtual memory systems.* We use this observation to help explain ideas and derive optimizations.

We explain Recursive ORAM through the example in Figure 4, which uses two levels of recursion. The system now contains 3 separate ORAM trees: the *Data ORAM*, denoted as ORam_0, and two *PosMap ORAMs*, denoted ORam_1 and ORam_2. Blocks in the PosMap ORAMs are akin to page tables. We say that PosMap blocks in ORam_i store X leaf labels for X blocks in ORam_{i-1}. This is akin to having X pointers to the next level page table. Generally, each PosMap level can have a different X. We assume the same X for all PosMaps for simplicity.

Suppose the LLC requests block a_0, stored in ORam_0. The leaf label l_0 for block a_0 is stored in PosMap block $a_1 = a_0/X$ of ORam_1 (division is floored throughout this paper). Like a page table, block a_1 stores leaves for neighboring data blocks (i.e., $\{a_0, a_0 + 1, \ldots, a_0 + X - 1\}$ in the case where a_0 is a multiple of X). The leaf l_1 for block a_1 is stored in the block

$a_2 = a_0/X^2$ stored in ORam_2. Finally, leaf l_2 for PosMap block a_2 is stored in the on-chip PosMap. The on-chip PosMap is now akin to the root page table, e.g., register CR3 on X86 systems.

To make a Data ORAM access, we must first lookup the on-chip PosMap, ORam_2 and ORam_1 in that order. Thus, a Recursive ORAM access is akin to a full page table walk.

Additional PosMap ORAMs ($\mathsf{ORam}_3, \ldots, \mathsf{ORam}_{H-1}$) may be added as needed to shrink the on-chip PosMap further. H denotes the total number of ORAMs (including the Data ORAM) in the recursion and $H = \log(N/p)/\log X + 1$ if p is the number of entries in the on-chip PosMap.

IV. Design Challenges for Hardware ORAM

The next two sections present the design of our hardware ORAM controller. This represents the first hardware ORAM with small client storage, integrity verification, or encryption units taped out and validated in silicon.

In this section, we first discuss major design challenges for an ORAM controller implemented in hardware. In the secure processor setting, the only prior hardware implementation of ORAM is an FPGA system called Phantom, by Maas et al. [30]. That work left several design challenges, which we address in Sections V and VI.

The first challenge is how to manage the position map (PosMap, Section III). The Phantom design does not use Recursive ORAMs. As a result, it requires multiple FPGAs *just to store the PosMap*, and thus is not suitable for integration with a single-chip secure processor.

We believe that to be practical and scalable to large ORAM capacities in secure hardware, Recursive ORAM (Section III-C) is necessary. Obviously, the trade-off is performance. One must access all the ORAMs in the recursion on each ORAM access. Counter-intuitively, with small block sizes, PosMap ORAMs can contribute to more than half of the total ORAM latency as shown in Figure 5. For a 4 GB Data ORAM capacity, 39% and 56% of bandwidth are spent on looking up PosMap ORAMs depending on the block size. Increasing the on-chip PosMap capacity only slightly dampens the effect. Abrupt kinks in the graph indicate when another PosMap ORAM is added (i.e., when H increases). In Section V-A, we show how insights from traditional virtual memory systems, coupled with security mechanisms, can dramatically reduce this PosMap ORAM overhead.

The second challenge in designing ORAM in hardware is how to maximize throughput. Ideally, we would like the limiting factor to be the memory bandwidth. Yet, the Phantom design showed that this was actually hard to achieve.

One bottleneck is the stash eviction logic (Step 5 in Section III-B). To decide where to evict blocks, Phantom constructs a hardware *heap sort* on the stash [30]. Unfortunately, this sorting step becomes the bottleneck under small block size and high memory bandwidth. For example, in the Phantom design, adding a block to the heap takes 11 cycles (see Appendix A of [30]). If the ORAM block size and memory bandwidth are such that accessing a block in memory takes less than 11 cycles, system performance is bottlenecked by the heap sort, not memory bandwidth. As a result, Phantom was parameterized with large block size, e.g., 4 KByte.

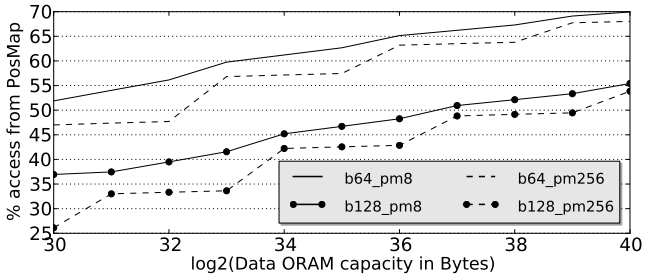


Fig. 5. The percentage of Bytes read from PosMap ORAMs in a full Recursive ORAM access for $X = 8$ (optimal found in [35]) and $Z = 4$. All bucket sizes are padded to 512 bits to estimate the effect in DDR3 DRAM. The notation b64_pm8 means the ORAM block size is 64 Bytes and the on-chip PosMap is at most 8 KB.

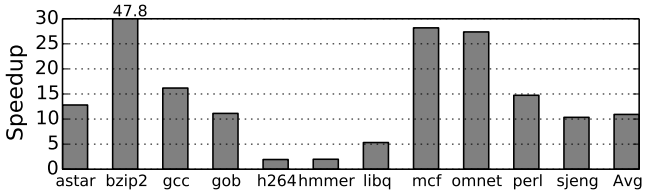


Fig. 6. The speedup achieved over a set of SPEC06 workloads by decreasing the block size from 4 KByte to 64 Bytes. The speedup assumes there is no bottleneck in the stash eviction logic or DRAM for any block size.

While benefiting applications with good data locality, a large block size severely hurts applications with erratic data locality. Figure 6 shows this effect. Motivated by the large potential speedup from small blocks, we will develop a stash eviction algorithm in Section VI-A that flexibly supports any practical block size (e.g., 64-Byte) without incurring performance loss.

Even after removing bottlenecks in the stash, using a small block size creates additional challenges in implementing ORAM over DRAM. Recall from Section III, accessing the ORAM requires the ORAM controller to walk down random paths in a binary tree where each node holds (say) $Z = 4$ blocks, *stored contiguously*. DRAM row locality in this operation is therefore based on Z and the block size, and shrinking the block size decreases this locality. Counter-intuitively, ignoring this issue can cause $2\times$ slowdown. In Section VI-B, we give a scheme that removes this bottleneck and allows ORAM to achieve $> 90\%$ of peak bandwidth.

V. ORAM Frontend

The techniques in this section only impact the Frontend and can be applied to any Position-based ORAM Backend (such as [38], [16], [34]). Section V-A presents a technique to optimize the PosMap. Section V-B discusses how to utilize the PosMap to implement integrity verification for ORAM.

A. PLB and Unified ORAM

1) PLB Caches and (In)security: Given our understanding of Recursive ORAM as a multi-level page table for ORAM (Section III-C), a natural optimization is to cache PosMap

blocks (i.e., page tables) so that LLC accesses exhibiting program address locality require fewer PosMap ORAM accesses on average. This idea is the essence of the *PosMap Lookaside Buffer*, or PLB, whose name obviously originates from the Translation Lookaside Buffer (TLB) in conventional systems.

Suppose the LLC requests block a_0. Recall from Section III-C that the PosMap block needed from ORam_i for a_0 has address $a_i = a_0/X^i$. If this PosMap block is in the PLB, the ORAM controller knows which path to access in ORam_{i-1}, and can skip ORam_i *and all the smaller PosMap ORAMs* ORam_j where $j > i$. Otherwise, block a_i is retrieved from ORam_i and added to the PLB. When block a_i is added to the PLB, another block may have to be evicted, in which case it is added to the stash of the corresponding PosMap ORAM.

Unfortunately, since PLB hits/misses correlate directly to a program's access pattern, the PosMap ORAM access sequence (filtered by PLB) leaks the program's access pattern (see [11] for a concrete example).

2) Security Fix: Unified ORAM Tree: To hide the PosMap access sequence, we will change Recursive ORAM such that *all PosMap ORAMs and the Data ORAM store blocks in the same physical tree* which we denote ORam_U. Organizationally, the PLB and on-chip PosMap become the new Path ORAM Frontend, which interacts with a single ORAM Backend.

Data blocks and the PosMap blocks originally from the PosMap ORAMs (i.e., $\mathsf{ORam}_1,\ldots,\mathsf{ORam}_{H-1}$) are now stored in a single ORAM tree (ORam_U) and all accesses are made to this one ORAM tree. Both data and PosMap blocks now have the same size. Since the number of blocks in ORam_i ($i > 0$) decreases exponentially with i, storing PosMap blocks alongside data blocks adds at most one level to the Unified ORAM tree ORam_U.

Each set of PosMap blocks must occupy a disjoint address space so that they can be disambiguated. For this purpose we apply the following addressing scheme: Given data block a_0, the address for the PosMap block originally in ORam_i for block a_0 is given by $i\|a_i$, where $a_i = a_0/X^i$. This address $i\|a_i$ is used to fetch the PosMap block from the ORam_U and to look up the PosMap block in the PLB. To simplify the notation, we don't show the concatenated address $i\|a_i$ in future sections and just call this block a_i.

Security-wise, both programs from the previous section access *only* ORam_U with the PLB and the adversary cannot tell them apart (see Section V-A5 for more discussion on security).

3) PLB Architecture: The PLB is a conventional hardware cache that stores PosMap blocks. Each PosMap block is tagged with its block address a_i and the current path it's mapped to. The path tag allows us to add an evicted PosMap block to the stash without accessing its own PosMap block. Simulation results show that increasing PLB size and associativity brings limited improvement [11], so we use an 8 KB direct-mapped PLB.

4) ORAM Access Algorithm: We introduce two new flavors of ORAM access to support PLB refills/evictions (i.e., two more types of *op* in Section III): read-remove and append. The idea of these two types of accesses appeared in [35] but we describe them in more detail below. Read-remove (readrmv) is the same as read except that it physically deletes the block from the stash after it is forwarded to the ORAM Frontend. Append (append) adds a block to the stash without performing an ORAM tree access. ORam_U must not contain duplicate blocks: only blocks that are currently not in the ORAM (possibly read-removed previously) can be appended. Further, when a block is appended, the current leaf it is mapped to in ORam_U must be known so that the block can be written back to the ORAM tree during later ORAM accesses.

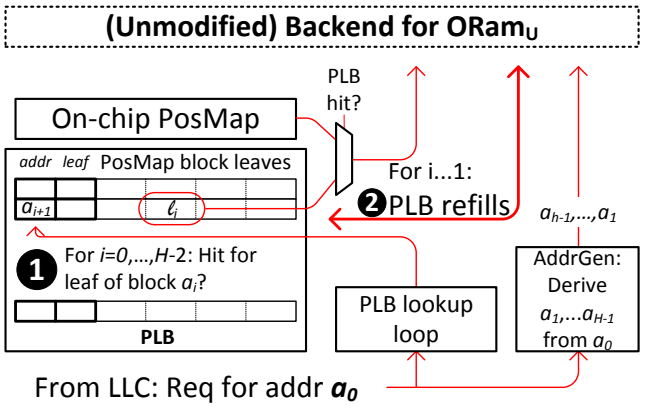


Fig. 7. PLB-enabled ORAM Frontend with $X = 4$. Accessing the actual data block a_0 (Step 3 in Section V-A4) is not shown.

The steps to read/write a data block with address a_0 are given below (shown pictorially in Figure 7):

1) **(PLB lookup)** For $i = 0,\ldots,H-2$, look up the PLB for the leaf of block a_i (contained in block a_{i+1}). If one access hits, save i and go to Step 2; else, continue. If no access hits for $i = 0,\ldots,H-2$, look up the on-chip PosMap for the leaf of block a_{H-1} and save $i = H-1$.
2) **(PosMap block accesses)** While $i \geq 1$, perform a readrmv operation to ORam_U for block a_i and add that block to the PLB. If this evicts another PosMap block from the PLB, append that block to the stash. Decrement i. (This loop will not be entered if $i = 0$.)
3) **(Data block access)** Perform an ordinary read or write access to ORam_U for block a_0.

Importantly, aside from adding support for readrmv and append, the above algorithm requires no change to the ORAM Backend.

5) Security Analysis: We now give a proof sketch that our PLB+Unified ORAM tree construction is secure. We make the following observations:

Observation 1. *If all leaf labels l_i used in* {read, write, readrmv} *calls to* Backend *are random and independent of each other, the* Backend *achieves the security of the original Path ORAM (Section III).*

Observation 2. *If an* append *is always preceded by a* readrmv*, stash overflow probability does not increase (since the net stash occupancy is unchanged after both operations).*

Theorem 1. *The PLB+Unified ORAM tree scheme reduces to the security of the ORAM* Backend.

Proof. The PLB+Unified ORAM Frontend calls Backend in two cases: First, if there is a PLB hit the Backend request is for

a PosMap or Data block. In this case, the leaf l sent to Backend was in a PosMap block stored in the PLB. Second, if all PLB lookups miss, the leaf l comes from the on-chip PosMap. In both cases, leaf l was remapped the instant the block was last accessed. We conclude that all {read, write, readrmv} commands to Backend are to random/independent leaves and Observation 1 applies. Further, an append command can only be caused by a PLB refill which is the result of a readrmv operation. Thus, Observation 2 applies. □

B. PosMap MAC for ORAM Integrity

The Ascend processor needs to additionally integrity-verify external memory. Now we describe a novel and simple integrity verification scheme for ORAM called *PosMap MAC*, or *PMMAC*. PMMAC achieves asymptotic improvements in hash bandwidth over prior schemes and is easy to implement in hardware.

It is well known that MAC is insufficient for memory integrity checking due to replay attacks. A fix for this problem is to embed a *non-repeating counter* in each MAC [37]. The challenge is how to make the counter *tamper-proof*. The idea of PMMAC is to use the *already existing* PosMap entries as tamper-proof non-repeating counters to facilitate the replay-resistant MAC scheme.

Suppose block a has data d and access counter c. We replace the PosMap entry for block a with c and generate the leaf l for block a as $l = \mathsf{PRF}_K(a \parallel c) \bmod 2^L$, where $\mathsf{PRF}_K()$ is a pseudorandom function [18], which we implement using AES-128. Block a is written to the Backend as the tuple (h, d) where

$$h = \mathsf{MAC}_K(c \parallel a \parallel d).$$

We implement $\mathsf{MAC}_K()$ using keyed SHA3-224. When block a is read, the Backend returns $(h^\star, d^\star)$ and PMMAC performs the following check to verify authenticity/freshness: assert $h^\star == \mathsf{MAC}_K(c \parallel a \parallel d^\star)$ where $\star$ denotes values that may have been tampered with. After the assertion is checked, c is incremented for the returned block.

Security follows if it is infeasible to tamper with block counters and no counter value for a given block is ever repeated. The first condition holds because the tamper-proof counters in the on-chip PosMap form the root of trust and then recursively, the PosMap blocks become the root of trust for the next level PosMap or Data ORAM blocks. The second condition can be satisfied by making each counter wide enough to not overflow, e.g., 64 bits wide.[5]

PMMAC significantly reduces the required hash bandwidth. A scheme based on Merkle tree checks and updates every hash on the path [33]. PMMAC only needs to check and update one block (the block of interest) per access, achieving an asymptotic reduction in hash bandwidth. For $Z = 4$ and $L = 16$, PMMAC reduces hash bandwidth by $Z(L + 1) = 68\times$.

[5]This causes the PosMap size to grow, since each entry in the original PosMap was $L + 1$ bits where $L < 32$ typically. As a result, we incur one additional level of recursion. In [11], we describe an optimization that compresses the 64 bit counters to be $< L$ bits. But we do not build it in hardware due to its extra complexity.

PMMAC requires no change to the ORAM Backend because the MAC is treated as extra bits appended to the original data block. The extra storage overhead is relatively small: the ORAM block size is usually 64-128 Bytes and a MAC is 80-128 bits.

1) Security Analysis: We first show that breaking our integrity verification scheme is as hard as breaking the underlying MAC. We start with the following observation:

Observation 3. *If the first $k - 1$ address and counter pairs (a_i, c_i)'s the* Frontend *receives have not been tampered with, then the* Frontend *seeds a MAC using a unique (a_k, c_k), i.e., $(a_i, c_i) \neq (a_k, c_k)$ for $1 \leq i < k$. This further implies $(a_i, c_i) \neq (a_j, c_j)$ for all $1 \leq i < j \leq k$.*

This property can be seen directly from the algorithm description. For every a, we have a dedicated counter, sourced from the on-chip PosMap or the PLB, that increments on each access.

Theorem 2. *Breaking the PMMAC scheme is as hard as breaking the underlying MAC scheme.*

Proof. We proceed via induction on the number of accesses. In the first ORAM access, the Frontend uses (a_1, c_1), to call Backend for (h_1, d_1) where $h_1 = \mathsf{MAC}_K(c_1 \parallel a_1 \parallel d_1)$. Note that a_1 and c_1 cannot be tampered with since they come from the Frontend. Thus, producing a forgery (h'_1, d'_1) where $d'_1 \neq d_1$ and $h'_1 = \mathsf{MAC}_K(c_1 \parallel a_1 \parallel d'_1)$ is as hard as breaking the underlying MAC. Suppose no integrity violation has happened and Theorem 2 holds up to access $n - 1$. Then, the Frontend sees fresh and authentic (a_i, c_i)'s for $1 \leq i \leq n - 1$. By Observation 3, (a_n, c_n) will be unique and $(a_i, c_i) \neq (a_j, c_j)$ for all $1 \leq i < j \leq n$. This means the adversary cannot perform a replay attack because all (a_i, c_i)'s are distinct from each other and are tamper-proof. It is also hard to generate a valid MAC with unauthentic data. Being able to produce a forgery (h'_i, d'_i) where $d'_i \neq d_i$ and $h'_i = \mathsf{MAC}_K(c_i \parallel a_i \parallel d'_i)$ means the adversary can break the underlying MAC. □

Next, to achieve privacy under active adversaries, we require certain assumptions about how the ORAM implementation will possibly behave in the presence of tampered data.

Property 1. *An ORAM* Backend *access only reveals to the adversary (a) the leaf sent by the* Frontend *for that access and (b) a fixed amount of encrypted data to be written back to the ORAM tree.*

The above properties hold in our implementation. The Frontend receives tamper-proof responses (by Theorem 2) and therefore produces independent and random leaves. Further, the global counter encryption scheme (Section III-B) trivially guarantees that the data written back to memory gets a fresh pad. It is then straightforward to see that any memory request trace generated by the Backend is indistinguishable from other traces of the same length.

C. A Note on ORAM Length Leakage Outside Ascend

We remark that the two optimizations for ORAM Frontend (PLB in Section V-A and PMMAC in Section V-B) can affect

the total length of the ORAM sequence. This is not a concern in the context of Ascend because a program always runs for a user-defined time budget T. However, if these optimizations are applied outside Ascend, we refer readers to [11], [13] for a discussion on the threat model and potential defenses.

VI. ORAM Backend

We now present several mechanisms to improve the ORAM Backend's throughput and make sure memory bandwidth is the performance bottleneck. The techniques in this section only impact the Backend and can be applied with or without the optimizations from Section V.

A. Stash Eviction Logic

As mentioned in Section IV, deciding where to evict each block in the stash is a challenge for Path ORAM hardware designs. In this section, we propose a new stash eviction algorithm that takes a single cycle to evict a block and can be implemented efficiently in hardware. This *eliminates* the stash eviction overhead for any practical block size and memory bandwidth.

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 occ. Stash can be thought of as a single-ported SRAM that stores data blocks and their metadata. Once populated, occ$[i]$ 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[occ$[i]$] for $i = 0, \ldots, Z(L+1)-1$ to be encrypted and written to external memory.

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 occ to $\perp$, to indicate that the eviction path is initially empty. Occupied is an $L+1$ entry array that records the number of real blocks that have been added to each bucket so far.

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, and outputs which level on path l that block should get written back to. In Line 16, t_1 represents in which levels the paths $P(l)$ and $P(l')$ diverge. In Line 17, t_2 is a one-hot bus where the set bit indicates the *first* level where $P(l)$ and $P(l')$ diverge. Line 18 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 20 further excludes buckets that already contain Z blocks (due to previous calls to PushBack()). Finally, Lines 21-23 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.

In hardware, we further improve Algorithm 1. First, we add 2 pipeline stages after Lines 17 and 18 in the PushBack() circuit to improve clock frequency. An important subtlety is that we don't add pipeline stages between when Occupied is read and updated, so a new block can be sent to PushBack() every cycle. Second, as soon as the leaf for the ORAM access is determined, blocks already in the stash are sent to the PushBack() circuit "in the background". After cycle C, each block read on the path is sent to the PushBack() circuit as soon as it arrives from external memory.

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:      occ ← {⊥ for i = 0, ..., (L+1)Z − 1}
4:      Occupied ← {0 for i = 0, ..., L}
5:      for i ← 0 to C + LZ − 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:             occ[offset] ← i
11:             Occupied[level] ← Occupied[level] + 1
12:         end if
13:     end for
14: end function
15: function PushBack(l, l', Occupied)
16:     t_1 ← (l ⊕ l') || 0                          ▷ Bitwise XOR
17:     t_2 ← t_1 & −t_1                  ▷ Bitwise AND, 2C negation
18:     t_3 ← t_2 − 1                              ▷ 2C subtraction
19:     full ← {(Occupied[i] ?= Z) for i = 0 to L}
20:     t_4 ← t_3 & ~full                  ▷ Bitwise AND/negation
21:     t_5 ← reverse(t_4)                        ▷ Bitwise reverse
22:     t_6 ← t_5 & −t_5
23:     t_7 ← reverse(t_6)
24:     if t_7 ?= 0 then
25:         return −1                      ▷ Block is stuck in stash
26:     end if
27:     return log_2(t_7)          ▷ Note: t_7 must be one-hot
28: end function
```

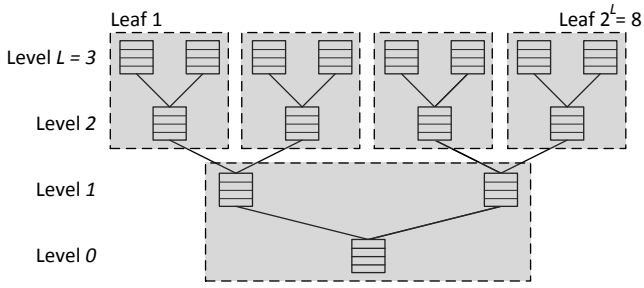


Fig. 8. Illustration of subtree locality.

B. Subtree Locality: Building Tree ORAMs on DRAM

Recall from Section IV: to fully reap the performance benefits of small blocks, we must address how to achieve high memory throughput for Path ORAM when implemented over DRAM. DRAM depends on spatial locality to offer high throughput: bad spatial locality means more DRAM row buffer misses which means time delay between consecutive accesses (we assume an open page policy on DRAM). However, when naïvely storing the Path ORAM tree into an array, two consecutive buckets along the same path hardly have any locality, and it can be expected that row buffer hit rate would be low.

To achieve high memory throughput for tree-based ORAMs,

we pack each subtree of k levels together, and treat them as the nodes of a new tree, a 2^k-ary tree with $\lceil \frac{L+1}{k} \rceil$ levels. Figure 8 is an example with $k = 2$. We set the node size of the new tree to be the row buffer size times the number of channels, which together with the original bucket size determines k. We adopt the address mapping scheme in which adjacent addresses first differ in channels, then columns, then banks, and lastly rows. With commercial DRAM DIMMs, $k = 6$ or $k = 7$ is possible which allows the ORAM to maintain $90 - 95\%$ of peak possible DRAM bandwidth. More details can be found in [35].

VII. ASIC Implementation and Measurements

We now evaluate a complete Ascend secure processor prototype in silicon, which was taped out March 2015 in 32 nm SOI, and was successfully tested in January 2017.

A. Chip Organization

The chip, shown in Figure 9, is composed of 25 cache-coherent SPARC T1 cores, an on-chip network, and the ORAM controller. The ORAM controller serves as the on-chip memory controller, intercepting LLC misses from the cores. The chip was done in collaboration with the Princeton OpenPiton project [3]. The Princeton team contributed the SPARC T1 cores and the on-chip network. We remark that the ORAM controller could have been connected to any cache/core hierarchy.

To implement Ascend, we require the (integrity-checked) ORAM controller (Sections V and VI), logic for timing protection and logic to initiate/terminate the server-user protocol (Section II). Since the ORAM controller already requires AES and SHA units, we simply reuse those existing components to perform the server-user protocol. To achieve timing channel protection, we did not explicitly implement the counter and queue as described in Section II-C but note that this logic requires negligible area.

B. Implementation Details

The ORAM controller was taped out with $L = 23$ and $B = 512$ bits. The entire design required five SRAM/RF memories (which we manually placed during layout): the PLB data array, PLB tag array, on-chip PosMap, stash data array and stash tag array. Numerous other (small) buffers were needed and implemented in standard cells. For PMMAC, we use flat 64 bit counters to check freshness. Thus, each PosMap block contains 8 counters, and we need six levels of recursion to achieve a final on-chip PosMap size of 8 KBytes.

We adopt AES and SHA units from OpenCores [1]. We use "tiny AES," a pipelined AES-128 core for memory encryption/decryption. Tiny AES has a 21 cycle latency and produces 128 bits of output per cycle. Two copies of a non-pipelined 12-cycle AES core are used as pseudorandom number generators, one in Frontend to generate new leaf labels, and the other in Backend to generate random paths for dummy accesses. We could use a single AES core for both purposes to save area, but opt for two separate cores for simplicity. A non-pipelined SHA3-224 core is used to implement $\mathsf{MAC}_K()$ for PMMAC. We truncate each MAC to 128 bits.

C. Tape-out Area and Performance

Post synthesis, the ORAM controller (which includes all hardware security components in Ascend) had a total area of 0.326 mm^2. For layout, we adopted a hierarchical work flow. We divided our ORAM controller into three logical modules: ORAM Frontend, ORAM Backend, and AES units. We placed and routed the three modules separately. Their respective dimensions and post-layout areas are given in Table II.

The bounding box of the ORAM controller was set to be 2 mm $\times$ 0.5 mm. This is due to an early design decision to put ORAM at the top edge of the chip as well as artificial constraints imposed by SRAM dimensions. Therefore, while the bounding box occupies $\sim$1 mm^2 area, the ORAM controller post-layout area is more accurately represented by the combined post-layout area of the three modules, which sum to $\sim$0.51 mm^2.

TABLE II
Dimensions (width $\times$ height) and area of the three modules of the ORAM controller.

Module	Frontend	Backend	Encryption
Dimensions (μm)	636.7 $\times$ 218.7	346.6 $\times$ 364.5	669.0 $\times$ 364.5
Area (mm^2)	0.139	0.126	0.244

Our design met timing at 1 GHz during place-and-route and can complete an ORAM access for 512 bits of user data (one cache line) in $\sim$ 1275 cycles (not including the initial round-trip delay to retrieve the first word of data from external memory). In an earlier work based on simulation results [11], we reported a very similar ORAM latency of 1208 cycles per access, which leads to an average slowdown of $\sim 4\times$ on SPEC-Int-2006 benchmarks with a typical cache hierarchy.

D. Functional Tests and Power Measurements in Silicon

We test ORAM functionality and measure its power consumption on first silicon. Table III shows the ORAM controller's dynamic power consumption across a range of voltages and clock frequencies. Dynamic power includes transistor switching power for the ORAM controller's logic, the SRAMs and the clock. For each test, core voltage (VDD) is set to the values shown in the table and SRAM voltage is set to 0.05 V higher than VDD. The power numbers do not include the power consumption from non-ORAM logic on the chip, the I/O pins or the external memory.

To measure peak power consumption, we need to keep the ORAM controller busy servicing memory requests at its highest throughput possible. Therefore, during power tests, we feed the ORAM controller with a synthetic memory request trace from an on-chip traffic generator, and use an on-chip buffer mimicking an external memory that has zero latency and can fully utilize the chip pin bandwidth. Thus, each measurement gives an upper bound on the chip power consumption in a real deployment.

In each test, we sample the chip current draw 100 times in 16 seconds and compute the average power. It is worth noting that the ORAM power consumption will gradually increase with time as the chip temperature increases. At lower voltage

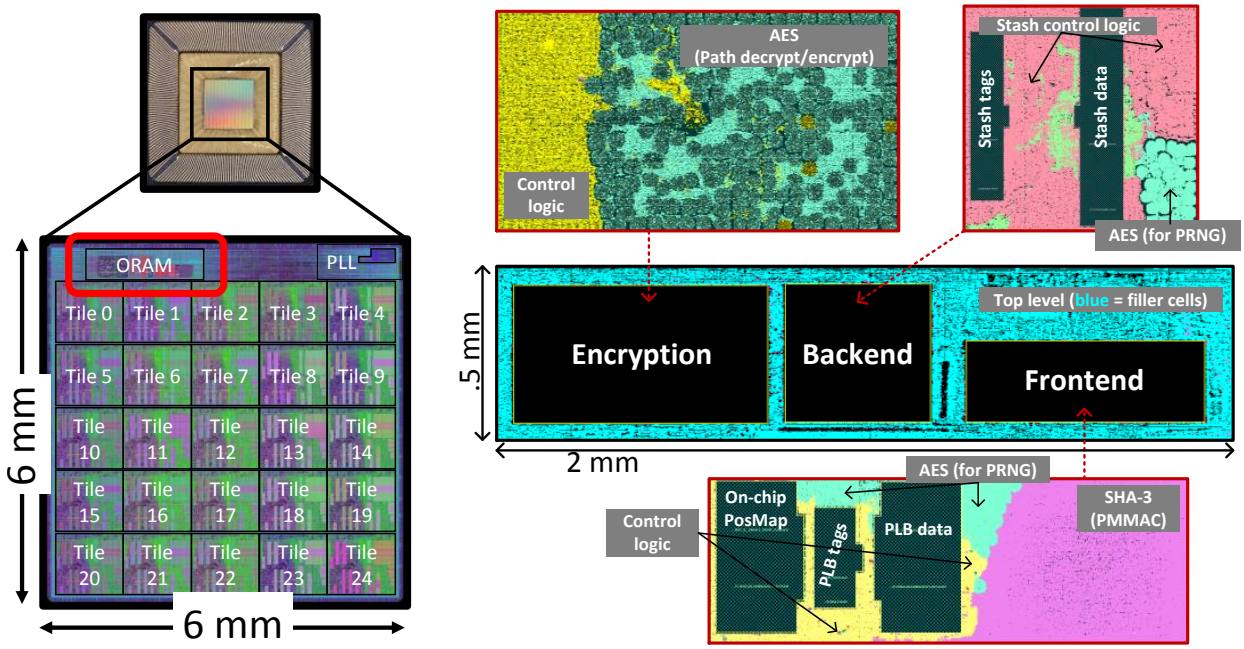


Fig. 9. Chip die photo (top left), whole-chip tape-out diagram (bottom left), and the ORAM controller broken up into the three logical modules (right).

TABLE III
ORAM CONTROLLER POWER CONSUMPTION (MW) UNDER DIFFERENT FREQUENCIES AND VOLTAGES.

V \ MHz	250	500	750	857
0.7	29.5			
0.75	32.4			
0.8	36.8			
0.85	43.2	74.8		
0.9	50.7	84.9		
0.95	57.6	97.9		
1.0			150	
1.1			208	299

(< 1 V), the effect of temperature increase is not noticeable and we start sampling the current 5 seconds after power on. At high voltage (≥ 1 V), this effect cannot be ignored, and we wait for the current draw to stabilize before sampling current.

Generally, running at a higher clock frequency requires a higher voltage to make transistors toggle faster. For each frequency in Table III, the ORAM logic will stop functioning (not meet timing) below a certain voltage, at which point we stop measuring power. For each frequency, the ideal point to run the ORAM controller is the lowest recorded voltage, which is the point that ORAM functions and consumes the least power. Since increasing voltage beyond the threshold strictly consumes more power, we omit the 1 V and 1.1 V measurements for frequencies 250 MHz and 500 MHz. Our test setup constrained us to test voltages ≤1.1 V. This is why we were only able to test frequencies up to 857 MHz. If equipped with a more effective cooling solution, the chip may function beyond 857 MHz with >1.1 V voltage. We repeat the test at 500 MHz and 0.9 V across three different chips. Dynamic power consumptions across chips vary by about 7%.

We also measure the power consumption from the clock tree. For these tests, the ORAM controller receives the clock and is ready to service memory requests but no memory request is made. For the frequencies and voltages tested in Table III, the clock tree accounts for around 40% of the total dynamic power.

VIII. Related Work

Academic work on single-chip (tamper-resistant) secure processors include eXecute Only Memory (XOM) [24], [25], [26], Aegis [41], [42] and Bastion [6]. In XOM, applications (both instructions and data) are only decrypted in secure compartments. XOM does not manage transparent spilling of data to a larger storage (e.g., cache misses to an external memory). Aegis, a single-chip secure processor, performs memory integrity verification and encryption on all data written to main memory, but does not provide access pattern or timing protection. Bastion provides the same external memory protection as Aegis, and uses a trusted hypervisor to protect applications when running alongside an untrusted operating system.

In the industry, secure processor extensions include ARM TrustZone [2], TPM+TXT [20] and most recently Intel SGX [22], [7]. Trustzone creates a "secure world" which isolates applications as long as they only require on-chip SRAM memory. TPM+TXT gives the user ownership over an entire machine, but does not provide encryption or other protection to main memory. Intel SGX (similar to Bastion) isolates applications from an untrusted operating system using hardware-supported enclaves, and provides encryption/integrity checks over data written to main memory.

None of the above works includes memory access pattern attacks or timing attacks in their threat model. An early work

that considers access pattern attacks is HIDE [48]. Using random shuffles for small chunks of memory, HIDE mitigates but does not completely stop information leakage from memory access patterns. Ascend is the first full system architecture that provides cryptographic security against an adversary that has complete control/visibility over external memory. More generally, Ascend prevents *untrusted* applications (buggy or arbitrarily malicious) from revealing user secrets outside the trusted chip through any digital side-channel. A concurrent hardware ORAM project named Phantom [30] is treated as a baseline design in our paper.

There have been a few works that propose additional techniques to improve hardware ORAM in the secure processor setting building on top of our work [47], [15]. They have adopted most of the techniques in this work such as the PLB and subtree locality. Nayak et al. has adopted our ORAM controller design in a secure hardware prototype for obfuscation [32]. Liu et al. present compiler techniques to reduce the required number of ORAM accesses [28] and evaluate on the Phantom system [27].

Outside the secure processor setting, ORAM has also found applications in various areas including storage outsourcing [39], [44], [9], [10], searchable encryption [36], secure computation [43], [29], proof of retrievability [5] and garbled RAM [17].

IX. Conclusion

This paper has described the Ascend execution model, for running *untrusted* programs operating safely on sensitive user data, as well as detailed implementation and measurement results for the Ascend prototype chip in silicon. This work proves the viability of a single-chip secure processor which can protect the privacy of software intellectual property or user data, as it interacts with an external memory device. The evaluation results are encouraging. The hardware mechanisms needed to support Ascend, when integrated into the 25 core test chip, are roughly half the size of a single processor core. Further, average program slowdown considering these mechanisms is estimated to be $\sim 4\times$ — roughly the cost of running a program in an interpreted language.

The Ascend execution model in its current form is somewhat constrained. Ascend does not support multiple tenants sharing the same chip, since on-chip resource sharing can leak private information. Other modules cannot write to Ascend main memory using DMA, and Ascend cannot be used in a multi-socket shared memory architecture. We leave these challenges to future work, and note that there have been efforts in these directions [22], [8], [27].

References

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

[2] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. In *Information Quarterly*, 2004.

[3] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. Openpiton: An open source manycore research framework. *ACM SIGOPS Operating Systems Review*, 50(2):217–232, 2016.

[4] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, 1996.

[5] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. *Journal of Cryptology*, 30(1):22–57, 2017.

[6] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA*, 2010.

[7] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016.

[8] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, pages 857–874, Austin, TX, 2016.

[9] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *USENIX Security*, pages 749–764, 2014.

[10] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. TCC, 2016.

[11] Christopher Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based Oblivious RAM. In *ASPLOS*, 2015.

[12] Christopher Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. A low-latency, low-area hardware Oblivious RAM controller. In *FCCM*, 2015.

[13] Christopher Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, 2014.

[14] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[15] Naoki Fujieda, Ryo Yamauchi, and Shuichi Ichikawa. Last path caching: A simple way to remove redundant memory accesses of path oram. In *Computing and Networking, Fourth International Symposium on*, pages 347–353, 2016.

[16] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PETS*, 2013.

[17] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, 2014.

[18] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 1986.

[19] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. In *J. ACM*, 1996.

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

[21] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, pages 1353–1364, 2016.

[22] Intel. Software guard extensions programming reference. Intel, 2013.

[23] Mohammad Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[24] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[25] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, pages 178–192, 2003.

[26] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS-IX*, 2000.

[27] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[28] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient RAM-model secure computation. In *Oakland*, 2014.

[29] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *SP*, pages 359–376, 2015.

[30] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[31] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW*, 2011.

[32] Kartik Nayak, Christopher Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. Hop: Hardware makes obfuscation practical. In *NDSS*, 2017.

[33] Ling Ren, Christopher Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity verification for Path Oblivious-RAM. In *HPEC*, 2013.

[34] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to Oblivious RAM. In *USENIX security*, 2015.

[35] Ling Ren, Xiangyao Yu, Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of Path Oblivious RAM in secure processors. In *ISCA*, 2013.

[36] Panagiotis Rizomiliotis and Stefanos Gritzalis. Oram based forward privacy preserving dynamic searchable symmetric encryption schemes. In *CCSW*, pages 65–76, 2015.

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

[38] 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.

[39] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *S&P*, 2013.

[40] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple Oblivious RAM protocol. In *CCS*, 2013.

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

[42] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *ISCA*. ACM, 2005.

[43] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. Cryptology ePrint Archive, Report 2014/672.

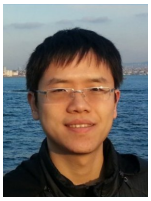
[44] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.

[45] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Oakland*, pages 640–656. IEEE, 2015.

[46] Xiangyao Yu, Christopher W Fletcher, Ling Ren, Marten van Dijk, and Srinivas Devadas. Generalized external interaction with tamper-resistant hardware with bounded information leakage. In *CCSW*, 2013.

[47] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. Fork path: improving efficiency of oram by removing redundant memory accesses. In *MICRO*, pages 102–114, 2015.

[48] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: An infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS*, 2004.



Ling Ren Ling Ren holds a Masters in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology and a Bachelors in Electrical Engineering from Tsinghua University, China. His research interests are in computer security, applied cryptography and computer architecture. While at MIT, he has worked on secure processors, ORAMs, and cryptocurrencies.

Christopher Fletcher Christopher Fletcher holds an S.M. and Ph.D. in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology and a B.S. in Electrical Engineering and Computer Science from the University of California, Berkeley. He will join the University of Illinois, Urbana-Champaign in 2017 as an Assistant Professor in the Computer Science Department. His research interests are in computer architecture, high-performance computing, computer security and applied cryptography.

Albert Kwon Albert Kwon holds a Masters in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology and a Bachelors of Science in Engineering in Electrical Engineering and Computer Science from University of Pennsylvania. His research interests are in security and privacy, applied cryptography, and distributed systems. He has worked on ORAMs and anonymous communication systems at MIT.

Marten van Dijk Marten van Dijk is an Associate Professor at the University of Connecticut. His research interests are in computer security and cryptography. He joined UConn in 2013. Prior to joining UConn, He worked at MIT CSAIL, RSA and Philips Research. He received a Ph.D. in mathematics, a M.S. in mathematics, and a M.S. in computer science from Eindhoven University of Technology.

Srinivas Devadas Srinivas Devadas is the Webster Professor of Electrical Engineering and Computer Science (EECS) at the Massachusetts Institute of Technology (MIT), where he has been since 1988. He received his MS and PhD from the University of California, Berkeley in 1986 and 1988, respectively. He served as Associate Head of EECS from 2005 to 2011. His research interests include Computer-Aided Design, computer architecture and computer security. He is a Fellow of the IEEE and ACM.