Integrity Verification for Path Oblivious-RAM

Ling Ren, Christopher W. Fletcher , Xiangyao Yu, Marten van Dijk and Srinivas Devadas
MIT CSAIL, Cambridge, MA, USA
{renling, cwfletch, yxy, marten, devadas}@mit.edu

Abstract—**Oblivious-RAMs (ORAM) are used to hide memory access patterns. Path ORAM has gained popularity due to its efficiency and simplicity. In this paper, we propose an efficient integrity verification layer for Path ORAM, which only imposes 17% latency overhead. We also show that integrity verification is vital to maintaining privacy for recursive Path ORAMs under active adversaries.**

I. INTRODUCTION

Privacy and integrity are two huge concerns in cloud storage and cloud computing. Privacy requires that an adversary not learn any information about the user's sensitive data. Integrity in storage outsourcing requires that the user's data is not tampered with; and in computation outsourcing it requires that the correct program is correctly executed on the correct input as the user requests.

In both applications, even if the data is encrypted, access patterns can leak a significant amount of information [1], [2]. Goldreich and Ostrovsky proposed Oblivious-RAM (ORAM) algorithms [3] to hide the memory access pattern. There has been significant follow-up work that has resulted in more efficient ORAM schemes [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. In this paper, we focus on a recent construction called Path ORAM [14], [15] for its simplicity and efficiency.

An interesting application of ORAM is to improve the security level of secure processors in cloud computing. Most previous secure processors—including Intel TXT [16], eXecute Only Memory (XOM) [17], [18], [19] and Aegis [20], [21]—do not prevent leakage from memory access patterns. Ascend [22] is a recently-proposed secure processor that uses recursive Path ORAM to conceal the memory access pattern. Ascend as described in [22] assumes only passive adversaries monitoring the Path ORAM and the pin traffic, and therefore cannot provide integrity of its computation.

In this paper, we show that recursive Path ORAMs without integrity verification cannot maintain privacy under active adversaries. Then we present an efficient integrity verification scheme for (recursive) Path ORAM, which adds only 17% latency overhead. With our mechanism and certified execution as described in [23], we can achieve both privacy and integrity in cloud computing by only trusting Ascend hardware and not the program or the server.

The rest of the paper is organized as follows: Section II reviews Path ORAMs. Section III examines Path ORAM security under active adversaries. Section IV and V present our integrity verification scheme for single and recursive Path ORAMs. We evaluate our proposed scheme in Section VI. Extentions and related work are discussed in Section VII. Section VIII concludes the paper.

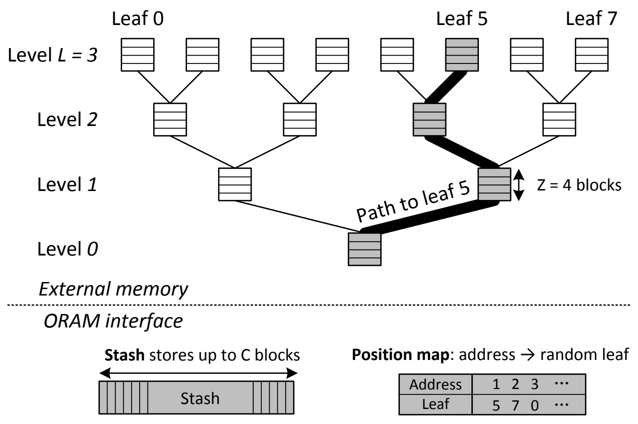

Fig. 1. A Path ORAM for $L = 3$ levels. Numbers indicate the steps (from Section II) to access a block mapped to leaf 5.

II. PATH ORAM

We briefly decribe Path ORAM below. Readers can refer to [15] for details.

Path ORAM organizes the physical external memory as a binary tree (Figure 1), where each node is a *bucket* that can hold up to Z data blocks. The root of the tree is referred to as level 0, and the leaves as level L. All the blocks are encrypted with probabilistic encryption. If a bucket has less than Z blocks, the remaining space is filled with encrypted dummy blocks.

A client makes requests to Path ORAM via a trusted *ORAM interface*. A client is the user in the cloud storage scenario, and is the secure processor in cloud computing. The ORAM interface is made up of two main structures, a *position map* and a *stash*. The position map is a lookup table that associates each data block with a leaf in the ORAM tree. The stash is a memory that can hold a small number of data blocks at a time. The external ORAM tree stores an (address, data, leaf) triplet for each block (with address 0 reserved for dummy blocks).

Path ORAM invariant. At any time, each data block in Path ORAM is uniformly randomly mapped to one of the 2^L leaves in the ORAM tree via the position map; if a block b is currently mapped to leaf l, then b must be stored either (i) on the path from the root to leaf l, or (ii) in the stash (see Figure 1). We also refer to the path from the root to leaf l as path l.

Path ORAM operations. When the client requests a block b with address u, the ORAM interface performs the following operations:

1) Look up the position map with u, yielding leaf label l.
2) Read all the buckets along path l. Decrypt all blocks and add all real blocks to the stash.
3) Return b on a read or update b on a write.

4) Replace l with a new random leaf l' (update the position map).
5) Evict (after probabilistic encryption) as many blocks as possible from the stash to path l. Fill any remaining space on the path with dummy blocks.

We will refer to Step 2–5 as $\mathsf{access}(\mathsf{ORam}, l)$. The path read and write (step 2 and 5) should be done in a data-independent way (e.g., from the root to the leaf).

Due to probabilistic encryption, the ciphertexts of all the blocks on path l will change. The only information revealed to an observer on $\mathsf{access}(\mathsf{ORam}, l)$ is the leaf label l. Step 4 guarantees that whenever a block is accessed, the old leaf label l is replaced by a new random leaf l'. So on each access, an observer only sees that a random path is read and written regardless of the address sequence actually requested.

Recursive Path ORAMs. The position map is usually too large, especially for a processor's on-chip storage. Recursive Path ORAMs can address the problem, where we store the original position map of ORam_0 in a second ORAM ORam_1. This trick can be repeated with however many ORAMs needed. We call ORam_0 data ORAM and all the other ORAMs position map ORAMs.

To be concrete, we give an example with a 2-level recursive ORAM $\{\mathsf{ORam}_0, \mathsf{ORam}_1\}$. Suppose each block in ORam_1 stores k entries of ORam_0's position map. The steps to access a block b with address u_0 are as follows:

1) Look up ORam_1's position map with $u_1 = \lfloor u_0/k \rfloor + 1$, yielding leaf label l_1.
2) Perform $\mathsf{access}(\mathsf{ORam}_1, l_1)$, yielding leaf label l_0. Update l_0 to a new random label l'_0 in ORam_1.
3) Perform $\mathsf{access}(\mathsf{ORam}_0, l_0)$ to obtain block b. During the operation update the leaf label in b's triplet to l'_0.

III. Path ORAM Security under Active Adversaries

Before describing our integrity verification scheme, we first point out that integrity verification is vital not only to the correct output of the ORAM interface, but also to the security of recursive Path ORAMs under active adversaries.

Specifically, we show how an active adversary can launch a replay attack by tampering with the Path ORAM and forcing the ORAM interface to behave differently under the following two access patterns: (a) repeatedly accessing the same data block, and (b) accessing different data blocks.

We use a 2-level recursive ORAM $\{\mathsf{ORam}_0, \mathsf{ORam}_1\}$ as an example. The adversary first initializes l^*_0, l^*_1 and $\mathcal{P}$ to $\perp$. Then for each ORAM access:

1) The ORAM interface performs $\mathsf{access}(\mathsf{ORam}_1, l_1)$ first. If $l_1 = l^*_1$, go to step 3); otherwise go to step 2).
2) Set $\mathcal{P}$ to be path l_1 in ORam_1 and $l^*_1 = l_1$. Carry out $\mathsf{access}(\mathsf{ORam}_1, l_1)$ honestly. When the ORAM interface performs $\mathsf{access}(\mathsf{ORam}_0, l_0)$, set $l^*_0 = l_0$ and act honestly. Wait for the next ORAM access and go back to step 1).
3) Return $\mathcal{P}$ to the ORAM interface. The ORAM interface then performs $\mathsf{access}(\mathsf{ORam}_0, l'_0)$. If $l'_0 = l_0$, guess access pattern (a); otherwise, guess access pattern (b).

Let L_i be the height of the ORam_i tree ($i = 0, 1$). Given a transcript of random leaf labels, for any two consecutive labels, $l_1 = l^*_1$ happens with 2^{-L_1} probability. If the transcript is long (on the scale of 2^{L_1}), then with high probability the adversary can find $l_1 = l^*_1$ somewhere in the transcript. Then in step 3), the ORAM interface gets $\mathcal{P}$, the snapshot of path l_1 before the last path write-back operation (rather than the latest version of path l_1). In this case, if the same block is accessed, the ORAM interface must get the same leaf label $l'_0 = l^*_0$ from the snapshot. However, if a different block is accessed, $l'_0 = l^*_0$ happens with only 2^{-L_0} probability. Therefore, the adversary distinguishes memory access pattern (a) from (b) with non-negligible (actually quite high) probability.

A lesson from this attack is that integrity verification cannot be done in the background. In other words, the ORAM interface cannot speculatively access ORam_i before verifying the leaf label it gets from ORam_{i+1}. Security breaks when the ORAM interface exposes the replayed label (l'_0 in the attack).

A single Path ORAM (non-recursive) is not vulnerable to this attack because its position map is securely stored within the ORAM interface. Thus a random path, obtained from the on-chip position map, is read and written on each access.

IV. Integrity-Verifying a Single Path ORAM

Our goal is to verify that data retrieved from Path ORAM is *authentic*, i.e., it was produced by the ORAM interface, and *fresh*, i.e., it corresponds to the latest version the ORAM interface wrote.

A naïve solution is to store a Merkle tree [24] in external memory, where each bucket in the ORAM tree corresponds to a leaf of the Merkle tree. We note that this scheme would work with any kind of ORAM, and similar ideas are used in [25]. To verify a bucket, the ORAM interface loads its corresponding path and siblings in the Merkle tree and checks the consistency of all the hashes. This scheme has large overheads for Path ORAM, because all the $(L + 1)$ buckets on a path have to be verified on each ORAM access.

To reduce the integrity verification overhead, we exploit the fact that *the basic operation of both the Merkle tree and Path ORAM is reading/writing paths through their tree structures*. This idea was first introduced in our prior work [26]. We present an improved version here.

We create an authentication tree that has exactly the same structure as Path ORAM (shown mirrored in Figure 2). Let B_i be the i-th bucket in the ORAM tree, and h_i be the corresponding node in the authentication tree. At a high level, each hash in the authentication tree depends on its two children as well as the corresponding bucket in the Path ORAM tree, i.e., $h_i = \mathsf{hash}(B_i || h_{2i+1} || h_{2i+2})$. (Note B_{2i+1} and B_{2i+2} are the two children of bucket B_i). The root hash h_0 is stored inside the ORAM interface and cannot be modified by an adversary. Similiar to a Merkle tree, the security of the proposed scheme can be reduced to the collision resistance of the hash function.

However, initially both the ORAM tree and the authentication tree contain random bits due to the uninitialized memory state, so the hash will not match. One solution is to initialize the authentication tree at start time, which involves touching every node in both trees at least once. To avoid initialization, we add two one-bit *child valid flags* to each bucket B_i—labeled f^i_0 and f^i_1 and stored in external memory along with B_i (except f^0_0 and f^0_1)—representing whether or not the children

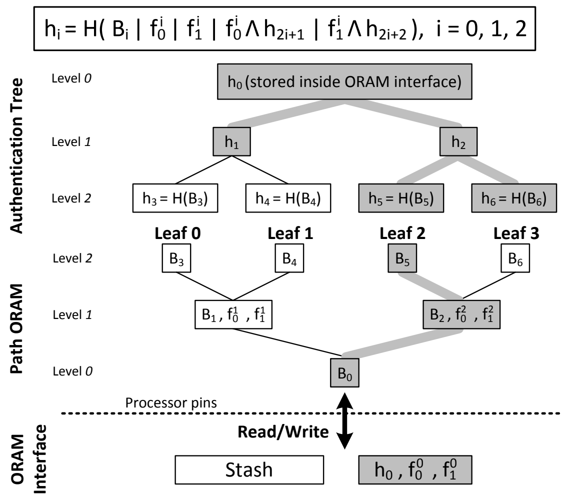

Fig. 2. Integrity verification of a single Path ORAM.

of bucket B_i have been initialized. Now we can exclude the uninitialized children from the hash input: if bucket B_i has been touched before, h_i should match

$$\mathsf{hash}(B_i||f_0^i||f_1^i||f_0^i \wedge h_{2i+1}||f_1^i \wedge h_{2i+2}) \qquad (1)$$

where '$\vee$' and '$\wedge$' are logical OR and AND operators, respectively. The leaf hashes only depend on the corresponding leaf buckets (they have no children or child valid flags).

We say a non-root bucket B_i is *reachable*, denoted as $\mathsf{reachable}(B_i) = 1$, if and only if the corresponding child valid flags equal 1 in all the buckets on the path from the root bucket to B_i. The root bucket B_0 is special since it has no parent. We let $\mathsf{reachable}(B_0) = f_0^0 \vee f_1^0$ and initialize $f_0^0 = f_1^0 = 0$. Conceptually, the child valid flags indicate a frontier in the ORAM/authentication trees that have been touched at an earlier time. We maintain the invariant that all reachable buckets have been written to in previous ORAM accesses, and their corresponding hashes in the authentication tree should match Equation (1). On each access, only the hash values for reachable buckets need to be checked.

Now we describe the steps of an ORAM access with the proposed integrity verification scheme. Following Figure 2, we give an example access to a block mapped to path $l = 2$. The ORAM interface performs the following operations:

1) ORAM path read: read buckets B_0, B_2 and B_5 into the stash, and child valid flags f_0^2, f_1^2.
2) Read the hashes on path $l = 2$ and the sibling hashes, i.e., h_1, h_2, h_5 and h_6.
3) Compute $\mathsf{reachable}(B_0) = f_0^0 \vee f_1^0$, $\mathsf{reachable}(B_2) = f_1^0$ and $\mathsf{reachable}(B_5) = f_1^0 \wedge f_0^2$. Let J be the set of reachable buckets on the path, i.e., $J = \{j | j \in \{0, 2, 5\}$ s.t. $\mathsf{reachable}(B_j) = 1\}$.
4) $\forall j \in J$, compute h'_j as in Equation 1.
5) If $\forall j \in J$, $h'_j = h_j$, the path is authentic and fresh!
6) Update child valid flags: $f_1^{2\prime} = f_1^2 \wedge \mathsf{reachable}(B_2)$, $f_0^{2\prime} = 1$, $f_1^{0\prime} = 1$ and $f_0^{0\prime} = f_0^0 \wedge \mathsf{reachable}(B_0)$.
7) ORAM path writeback: write back bucket B_0, B_2, B_5, and $f_0^{2\prime}, f_1^{2\prime}$ as the new child valid flags for B_2.
8) Re-compute h_5, h_2 and h_0; write back h_5 and h_2.

All data touched in external memory is shaded in Figure 2.

We emphasize that both the hashes along the path of interest and the sibling hashes should be read into the ORAM interface, so that in Step 4 the hashes can be computed in parallel. In summary, on each ORAM access $2L$ hashes (along the path and the sibling) need to be read into the ORAM interface and L hashes (along the path) need to be updated in the external authentication tree.

Suppose each bucket contains M bits of ciphertext (including the randomness used in probabilistic encryption). When building Path ORAM on DRAM, buckets are usually aligned to DRAM access granularity of D bits (a typical value of D is 64 bytes = 512 bits). Then, the remaining space in an aligned bucket is $Q = \lceil M/D \rceil \cdot D - M$. Suppose the hash is V bits long. The remaining space in a bucket can contain up to $t = \lfloor Q/V \rfloor$ hashes. The implementation depends on t:

1) If $t \geq 2$, we store with each bucket B_i its two child hashes (i.e., h_{2i+1} and h_{2i+2}). In this case, integrity verification does not add storage or communication overhead on top of Path ORAM.
2) If $t = 1$, we pack h_i along with each bucket B_i. On a path read, the sibling hashes need to be fetched into the ORAM interface, adding 1 extra read per bucket. On a path write-back, only the hashes on the path of interest are updated, without extra memory accesses.
3) If $t = 0$, we pack the child hashes with each bucket (as in the $t \geq 2$ case). This adds D bits storage per bucket, and 1 read and 1 write per access.

Obviously, $t \geq 2$ is the ideal situation. Given M, D, V we can tune parameters (e.g., the block size of position map ORAMs) to make $t = 2$ (see Section VI).

V. Integrity-Verifying Recursive Path ORAMs

A direct solution is, of course, to apply the above scheme for a single ORAM to every ORAM in the recursion. As noted before, integrity verification for position map ORAMs is on the critical path: the ORAM interface has to verify that all the hashes match before reading the next ORAM. The latency of evaluating a hash function (e.g., SHA-1) often depends on the length of the input. In this section, we show that the input to the hash can be significantly reduced for position map ORAMs, if we use a slightly relaxed but still strong enough security definition for integrity verification.

Definition 1. *An integrity verification scheme for ORAMs is secure, if no computationally bounded adversary with the ability to modify ORAMs can with non-negligible probability (a) change the output of the ORAM interface without being detected, or (b) learn anything about the memory access pattern.*

The above security definition is weaker than that of hash trees in the sense that it does not require detecting modifications to ORAMs that do not change the output of the ORAM interface. The definition is still sufficiently strong because under such modifications the client still receives the correct data from the ORAM interface.

Our main theorem relies on the use of *seeds* in probabilistic encryption schemes based on pseudorandom functions.

$$\mathsf{encrypt}_K(X) = (s, G_K(s) \oplus X)$$

where $G = \{G_K : \{0,1\}^{|s|} \to \{0,1\}^{|X|}\}$ is a family of pseudorandom functions indexed by key K. The length of the key K is the security parameter. To encrypt a message X, the encryption module feeds a seed s to G_K, and then XORs the output (as a one-time pad) to X. The ciphertext is the seed s together with the one-time padded string $Y = G_K(s) \oplus X$.

The pseudorandomness of G_K guarantees that $G_K(s)$ is indistinguishable from truly random strings. The security of the encryption also requires that the ORAM interface reuses the same seed with at most negligible probability (if the same seed is used in both $G_K(s) \oplus X_1$, $G_K(s) \oplus X_2$, then $X_1 \oplus X_2$ would be revealed to the adversary). This can be guaranteed by using $s = BucketCtr||BucketID$, where $BucketCtr$ is a 64-bit counter that is incremented on each access and is initialized to 0 when a bucket is first accessed.

Given Definition 1, we make the following key observation.

Theorem 1. *To get a secure integrity verification scheme for recursive Path ORAMs, it suffices to integrity-verify data ORAM and the seeds for position map ORAMs.*

Proof outline. The proof of the theorem is based on two insights. First, data ORAM stores (address, data, leaf) triplets. If the ORAM interface gets the wrong leaf label from corrupted position map ORAMs, it can detect that in data ORAM. Second, since the seeds s are integrity-verified, an adversary cannot steer the ORAM interface to re-using the same seed with the same key. If the seeds in position map ORAMs are fresh and authentic, any ciphertext produced by an adversary will decrypt into random bits. In that case, the ORAM interface will behave indistinguishably—always accesssing a random path in each ORAM—and the adversary cannot learn anything about the access pattern. We will prove these two insights as lemmas.

Consider a recursive Path ORAM with H levels $(\mathsf{ORam_i})_{i=0}^{H-1}$, where $\mathsf{ORam_0}$ is the data ORAM. From a client's view, the ORAM interface takes in a sequence of addresses $\mathbf{u} = (u_j)_{j=1}^m$, and outputs a sequence of blocks $\mathbf{b} = (b_j)_{j=1}^m$, where m is the total number of accesses, and is polynomial in the security parameter. From an observer's view, the ORAM interface exposes a leaf label (accesses a path) for each ORAM in the recursion. Let l_j^i be the path read and written for $\mathsf{ORam_i}$ on the j-th ORAM access, where $i \in \{0, 1, \cdots, H-1\}, j \in \{0, 1, \cdots, m\}$.

We can regard all the position map ORAMs combined as PosMap, which takes the address sequence $\mathbf{u}$ as input, and returns to the ORAM interface a sequence of leaf labels $\mathbf{l^0} = (l_j^0)_{j=1}^m$ for accessing the data ORAM. For convenience, we denote $\mathbf{l^0} = \mathsf{PosMap}(\mathbf{u})$ as the correct leaf labels returned by the original (unmodified) position map ORAMs, and $\mathbf{l^{0\prime}} = \mathsf{PosMap}'(\mathbf{u})$ as the leaf labels returned by the modified position map ORAMs.

Lemma 1. *Given* $\mathsf{ORam_0}$ *is authentic and fresh, if* $\exists j$ *where* PosMap' *yields* $l_j^{0\prime} \neq l_j^0$*, then the ORAM interface will detect this when accessing* $\mathsf{ORam_0}$.

Lemma 2. *Given the seeds are authentic and fresh, whichever way an adversary tampers with any* ORam_i*,* $1 \leq i \leq H-1$*, from the view of the adversary* $l_j^{i\prime}$ *is computationally indistinguishable from uniformly random for any* i, j.

Proof of Lemma 1: We show that the ORAM interface can detect the first time (the smallest j) that $l_j^{0\prime} \neq l_j^0$, which is sufficient to prove the lemma. Since data ORAM is authentic and fresh (it is integrity-verified using the method in Section IV), the Path ORAM invariant guarantees that a triplet (b_j, u_j, l_j^0) is stored somewhere along path l_j^0 in the $\mathsf{ORam_0}$ tree or in the stash. If due to the wrong output of PosMap', the ORAM interface performs $\mathsf{access}(\mathsf{ORam_0}, l_j^{0\prime})$, then either:

1) block b_j is not found along path $l_j^{0\prime}$ or the stash, and the ORAM interface knows $l_j^{0\prime}$ is wrong;
2) block b_j is found in the stash or on the common subpath of path $l_j^{0\prime}$ and path l_j^0, the ORAM interface compares $l_j^{0\prime}$ with the leaf label stored in the triplet and finds $l_j^{0\prime} \neq l_j^0$.

In either case, the ORAM interface detects that the position map ORAMs have been modified. ■

Proof of Lemma 2: First note that $l_j^{H-1\prime}$, the path read/written in the smallest ORAM, is stored securely inside the ORAM interface, so it is uniformly random.

Leaf labels for the other ORAMs, $l_j^{i\prime}$ $(i < H-1)$, are produced during the execution of PosMap'. Suppose the original leaf label l_j^i is part of a block X in $\mathsf{ORam_{i+1}}$. By $[X]$ we denote the restriction of X to the bits that represent the leaf labels for $\mathsf{ORam_i}$; this excludes the address and leaf for X itself in the triplet X. At any time, $[X]$ is uniformly random[1] because it always contains *fresh* leaf labels. (Recall that whenever a leaf label is used in ORAM operations, it will be replaced with a new random one). So the same restriction of the ciphertext $[Y] = [G_K(s) \oplus X] = [G_K(s)] \oplus [X]$ is also uniformly random, and statistically independent from $[G_K(s)]$. An adversary can change Y to Y'. Since Y' only depends on Y, $[Y']$ and $[G_K(s)]$ are also statistically independent. The seed s is authentic and fresh, so the ORAM interface decrypts Y' into $X' = G_K(s) \oplus Y'$. Now we have $G_K(s)$ is computationally indistinguishable from uniformly random strings and is statistically independent from $[Y']$, so the resulting $[X']$ and $l_i^{i\prime}$ (one of the leaf labels in $[X']$) are also indistinguishable from uniformly random strings. ■

Lemma 1 states that the ORAM interface can detect any wrong leaf label it gets from PosMap' when it accesses the authentic and fresh data ORAM. Note that Lemma 1 does not preclude the possibility that PosMap', though modified, still gives the correct leaf label l_j^0. We emphasize that the client will still get the authentic and fresh data from the ORAM interface in this case.

Lemma 2 states that as long as the seeds are verified, the paths accessed for each ORAM in the recursion by the ORAM interface will be computationally indistinguishable from uniformly random, regardless of how an adversary tampers with the rest of the contents in position map ORAMs. In the proof of Lemma 2, it is important that the seeds are verified. If s can be tampered with, then X' may not be $G_K(s) \oplus Y'$. The attack in Section III takes advantage of this security hole.

Combining the two lemmas completes the proof of Theorem 1 and the security of our proposed scheme.

[1]In practice, $[X]$ may be generated by some pseudorandom number generator. Then statistical independence in the proof becomes computational independence, and the lemma still holds.

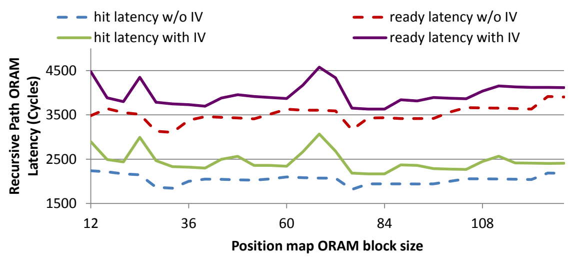

Fig. 3. Integrity verification overhead for 8GB recursive Path ORAMs using different position map ORAM block size. Each Path ORAM in the recursion uses $Z = 3$ and 50% dummy blocks.

VI. Evaluation

We evaluate our integrity verification assuming Path ORAM is implemented on commodity DRAM. We use DRAMSim2 [27], a publicly available, open source JEDEC DDRx memory system simulator, which models memory controllers/channels, DRAM banks/ranks. Assuming data ORAM is 8 GB with 50% utilization (resulting in 4 GB working set) and has a 128-byte block size, the best configuration in [26] uses $Z = 3$ and a 32-byte block size for position map ORAMs, which we take as the baseline. Following [26], we assume four independent channels and use DRAMSim2's default DDR3_micron configuration with 16-bit device width, 1024 columns per row in 8 banks, and 16384 rows per DRAM-chip. We also assume CPU frequency is $4\times$ that of DDR3. We use the SHA-1 hash (160 bits) and assume an 80-cycle latency of hashing 512 bits and a 50-cycle encryption/decryption latency.

[26] has explored the design space of recursive Path ORAMs without integrity verification. With integrity verification, the optimal configuration of recursive Path ORAMs differs from what's proposed in [26]. We can tune position map ORAM block size to make space for the two child hashes in each bucket. Figure 3 gives the latency overhead of integrity-verifying recursive Path ORAMs with different position map ORAM block sizes. Hit delay is the latency to return data (when path reads for all ORAMs and decryption finish); ready delay is the time until Path ORAM can be accessed again (when path write-back for all ORAMs finish). With integrity verification the optimal position map block size is 80 bytes. It increases hit delay by 17.4% and ready delay by 16.9%, compared with the baseline. If we continue using a 32-byte position map block size, integrity verification increases hit delay by 26.3% and ready delay by 20.7%.

Theorem 1 enables us to reduce the hash input for position map ORAMs, from the entire bucket to just a seed, besides two child hashes. If we use 128-bit seeds and SHA-1 hash, the reduced hash input is $128 + 2 \times 160 = 448$ bits, less than the 512 bit SHA-1 input chunk, so we pay only one SHA-1 latency regardless of the bucket size in position maps ORAMs. Without this optimization, 80-byte blocks with $Z = 3$ will form 240-byte buckets. Adding two child hashes, the hash input is at least 280 bytes (more than 4×512 bits). It would take at least 240 cycles to hash them. This would add $160 \times 2 = 320$ cycles (there are 2 position map ORAMs), which is a 15% additional latency overhead for hit delay and 9% for ready delay.

VII. Discussion and Related Work

A. Child Valid Flags

In our integrity verification scheme (Section IV) we propose using child valid flags to avoid initializing the authentication tree. This trick is also important for distinguishing real blocks from dummy blocks in the Path ORAM tree. Define Path ORAM utilization to be the ratio between the number of real blocks and the number $Z \cdot (2^{L+1} - 1)$ of total blocks in ORAM. Experiments in both [14] and [26] have shown that the utilization of Path ORAM should not be set too high. Otherwise we will need either a very large stash or lots of dummy accesses. However, if initially the external ORAM tree contains random bits, these random bit strings, interpreted as ciphertexts, may decrypt into legitimate-looking real blocks. So Path ORAM begins with an unpredictable utilization. This problem looms large when dummy blocks are just identified with a special address (e.g., all zero) as in [26]: in this case, almost all random ciphertexts will decrypt into real blocks.

The child valid flags can address this problem and avoid the need for initialization. We stored the child valid flags along with each bucket. When a path is accessed, the ORAM interface reads the path from the root towards the leaf. Before reading bucket B_i, it first computes $\mathsf{reachable}(B_i)$ in the same way as Step 3 in Section IV. As soon as it finds an unreachable bucket, the rest of the path must be unreachable and does not need to be read. The ORAM interface updates the child valid flags on the path write-back as it does in Section IV.

B. Extending Theorem 1 to Other Encryption schemes

In the proof of Lemma 2 and Theorem 1, we restrict the probabilistic encryption schemes to those based on pseudo-random functions and one-time pads. In practice, we can use AES counter mode. We can also extend our proof to any probabilistic encryption schemes that have the desired property: any ciphertexts produced by the adversary will decrypt into random bits. It requires an encryption scheme to have some verified randomness to defeat the replay attack in Section III. For example, we can use the following keyed AES:

$$\mathsf{encrypt}_K(X) = (AES_K(K'), AES_{K'}(X))$$

where K' is a random key (playing the role of a seed) and is verified.

C. Extending Theorem 1 to Background Evictions

For the original Path ORAM proposed in [14], [15], each ORAM in the recursion always returns results with a public and pre-determined latency and this is implicitly assumed in the proof of Theorem 1.

But Path ORAM has a probability of stash overflow. [15] proves that the overflow probability is negligible for $Z \geq 8$. To prevent stash overflow for small Z, which has better performance, [26] proposes background eviction. Background eviction slowly empties the stash using dummy accesses. It does not affect security since dummy accesses are indistinguishable from real accesses: both read/write random paths. However, background eviction introduces a timing channel to Path ORAM. Path ORAM now returns results with an uncertain latency, depending on the state of the stash.

With background eviction, Theorem 1 is still correct: it still guarantees that the ORAM interface yields the correct output and that an adversary cannot learn anything about the access pattern. But an active adversary may be able to 'slown down' Path ORAMs, if it can turn some dummy blocks into real blocks to increase ORAM utilization. A higher utilization will cause more dummy accesses and increase the average latency of Path ORAM when background eviction is used.

If we want to include "correct latency" in Definition 1, we can simply use a larger Z without background eviction. Another solution is to issue dummy accesses at a fixed rate, which eliminates the timing channel of Path ORAM. The advantage of this approach is that we can still use the aggressive settings if dummy rate is set appropriately; the downside is that we no longer have the guarantee that the stash will never overflow.

D. Related Work

Merkle trees (or hash trees) were proposed in [24] as a means to efficiently verify and update the signature of large documents. Aegis is the first secure processor to provide integrity verification for normal insecure memory (DRAM). It adopts the efficient memory integrity verification scheme proposed in [28], where the hash tree is cached. However, such optimizations were based on the locality in neighboring memory accesses, and therefore do not work for Path ORAM because Path ORAM shuffles the memory and always accesses random locations throughout the entire memory.

VIII. Conclusion

We have proposed an efficient integrity verification method for Path ORAM. Our integrity verification only increases Path ORAM latency by 17%.

References

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

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

[3] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," in *J. ACM*, 1996.

[4] R. Ostrovsky and V. Shoup, "Private information storage (extended abstract)," in *STOC*, 1997, pp. 294–303.

[5] R. Ostrovsky, "Efficient computation on oblivious rams," in *STOC*, 1990.

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

[7] 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.

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

[9] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious RAM and a new balancing scheme," in *SODA*, 2012.

[10] P. Williams and R. Sion, "Round-optimal access privacy on outsourced storage," in *CCS*, 2012.

[11] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Practical oblivious storage," in *CODASPY*, 2012.

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

[13] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *NDSS*, 2012.

[14] E. Stefanov and E. Shi, "Path O-RAM: An Extremely Simple Oblivious RAM Protocol," Cornell University Library, arXiv:1202.5150v1, 2012, arxiv.org/abs/1202.5150.

[15] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path O-RAM: An Extremely Simple Oblivious RAM Protocol," Cornell University Library, arXiv:1202.5150v2, 2013, arxiv.org/abs/1202.5150, to appear in CCS 2013.

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

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

[18] D. Lie, C. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 178–192.

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, November 2000, pp. 168–177.

[20] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, " AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *Proceedings of the* 17^{th} *ICS (MIT-CSAIL-CSG-Memo-474 is an updated version)*. New-York: ACM, June 2003. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-474/Memo-474.pdf (revised one)

[21] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions," in *Proceedings of the* 32^{nd} *ISCA'05*. New-York: ACM, June 2005. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-483/Memo-483.pdf

[22] C. Fletcher, M. van Dijk, and S. Devadas, "Secure Processor Architecture for Encrypted Computation on Untrusted Programs," in *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing*, Oct. 2012, pp. 3–8.

[23] C. W. Fletcher, "Ascend: An architecture for performing secure computation on encrypted data," in *MIT CSAIL CSG Technical Memo 508 (Master's thesis)*, April 2013. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-508/Memo-508.pdf

[24] R. C. Merkle, "Protocols for public key cryptography," in *IEEE Symposium on Security and Privacy*, 1980, pp. 122–134.

[25] J. R. Lorch, J. W. Mickens, B. Parno, M. Raykova, and J. Schiffman, "Toward practical private access to data centers via parallel oram." *IACR Cryptology ePrint Archive*, vol. 2012, p. 133, 2012, informal publication. [Online]. Available: http://dblp.uni-trier.de/db/journals/iacr/iacr2012.html#LorchMPRS12

[26] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *Proceedings of the Int'l Symposium on Computer Architecture*, June 2013, available at Cryptology ePrint Archive, Report 2012/76.

[27] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[28] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and Merkle Trees for Efficient Memory Integrity Verification," in *Proceedings of Ninth International Symposium on High Performance Computer Architecture*. New-York: IEEE, February 2003.