cryptography

Article

Flat ORAM: A Simplified Write-Only Oblivious RAM Construction for Secure Processors

Syed Kamran Haider * and Marten van Dijk

Department of Electrical and Computer Engineering, University of Connecticut, Storrs, CT 06269-4157, USA; marten.van_dijk@uconn.edu

* Correspondence: syed.haider@uconn.edu

Received: 22 January 2019; Accepted: 20 March 2019; Published: 25 March 2019

Abstract: Oblivious RAM (ORAM) is a cryptographic primitive which obfuscates the access patterns to a storage, thereby preventing privacy leakage. So far in the current literature, only 'fully functional' ORAMs are widely studied which can protect, at a cost of considerable performance penalty, against the strong adversaries who can monitor all read and write operations. However, recent research has shown that information can still be leaked even if only the write access pattern (not reads) is visible to the adversary. For such weaker adversaries, a fully functional ORAM turns out to be an overkill, causing unnecessary overheads. Instead, a simple 'write-only' ORAM is sufficient, and, more interestingly, is preferred as it can offer far better performance and energy efficiency than a fully functional ORAM. In this work, we present Flat ORAM: an efficient write-only ORAM scheme which outperforms the closest existing write-only ORAM called HIVE. HIVE suffers from performance bottlenecks while managing the memory occupancy information vital for correctness of the protocol. Flat ORAM introduces a simple idea of Occupancy Map (OccMap) to efficiently manage the memory occupancy information resulting in far better performance. Our simulation results show that, compared to HIVE, Flat ORAM offers 50% performance gain on average and up to 80% energy savings.

Keywords: cloud computing security; hardware security; secure computer architectures; oblivious ram; side channels

1. Introduction

User's data privacy concerns in computation outsourcing to cloud servers have gained serious attention over the past couple of decades. To address this challenge, various trusted-hardware based secure processor architectures have been proposed including TPM [1–3], Intel's TPM+TXT [4], eXecute Only Memory (XOM) [5–7], Aegis [8,9], Ascend [10], Phantom [11], and Intel's SGX [12]. A trusted-hardware platform receives user's encrypted data, which is decrypted and computed upon inside the trusted boundary, and finally the encrypted results of the computation are sent to the user.

Although all the data stored outside the secure processor's trusted boundary, e.g., in the main memory, can be encrypted, information can still be leaked to an adversary through the access patterns to the stored data [13]. In order to prevent such leakage, Oblivious RAM (ORAM) is a well-established technique, first proposed by Goldreich and Ostrovsky [14]. ORAM obfuscates the memory access pattern by introducing several randomized 'redundant' accesses, thereby preventing privacy leakage at a cost of significant performance penalty. Intense research over the last few decades has resulted in more and more efficient and secure ORAM schemes [15–28].

A key observation regarding the adversarial model assumed by the current renowned ORAM techniques is that the adversary is capable of learning fine-grained information of all the accesses made to the memory. This includes the information about which location is accessed, the type of operations (read/write), and the time of the access. It is an extremely strong adversarial model which, in most cases, requires direct physical access to the memory address bus in order to monitor both read and write accesses, e.g., the case of a *curious* cloud server.

On the other hand, for purely remote adversaries (where the cloud server itself is trusted), direct physical access to the memory address bus is not possible thereby preventing them from directly monitoring read/write access patterns. Such remote adversaries, although weaker than adversaries having physical access, can still "learn" the application's write access patterns. Interestingly, privacy leakage is still possible even if the adversary is able to infer just the write access patterns of an application.

John et al. demonstrated such an attack [29] on the famous Montgomery's ladder technique [30] commonly used for modular exponentiation in public key cryptography. In this attack, a 512-bit secret key is correctly inferred in just 3.5 min by only monitoring the application's write access pattern via a compromised Direct Memory Access (DMA) device on the system [31–35] (DMA is a standard performance feature which grants full access of the main memory to certain peripheral buses, e.g., FireWire, Thunderbolt etc.). The adversary collects the snapshots of the application's memory via the compromised DMA. Clearly, any two memory snapshots only differ in the locations where the data has been modified in the latter snapshot. In other words, comparing the snapshots not only reveals the fact that write accesses (if any) have been made to the memory, but it also reveals the exact locations of the accesses which leads to a precise access pattern of memory writes resulting in privacy leakage.

Recent work [36] demonstrated that DMA attacks can also be launched remotely by injecting malware to the dedicated hardware devices, such as graphic processors and network interface cards, attached to the host platform. This allows even a remote adversary to learn the application's write access pattern. Intel's TXT has also been a victim of DMA based attacks where a malicious OS directed a network card to access data in the protected VM [37,38].

One approach to prevent such attacks, as adapted by TXT, could be to block certain DMA accesses through modifications in DRAM controller. However, this requires the DRAM controller to be included in the trusted computing base (TCB) of the system which is undesirable. Nevertheless, there could potentially be many scenarios other than DMA based attacks where write access patterns can be learned by the adversary.

Current *full-featured* ORAM schemes, which obfuscate both read and write access patterns, also offer a solution to such weaker adversaries. However, the added protection (obfuscation of reads) offered by full-featured ORAMs is unnecessary and is practically an overkill in this scenario which results in significant performance penalty. Path ORAM [24], the most efficient and practical fully functional ORAM system for secure processors so far, still incurs about 2–10$\times$ performance degradation [10,39] over an insecure DRAM system. A far more efficient solution to this problem is a *write-only* ORAM scheme, which only obfuscates the write accesses made to the memory. Since read accesses leave no trace in the memory snapshot and hence do not need to be obfuscated in this model, a write-only ORAM can offer significant performance advantage over a full-featured ORAM.

A recent work, HIVE [40], has proposed a write-only ORAM scheme for implementing hidden volumes in hard disk drives. The key idea is similar to Path ORAM, i.e., each data block is written to a new random location, along with some dummy blocks, every time it is accessed. However, a fundamental challenge that arises in this scheme is to avoid *collisions*, i.e., to determine whether a randomly chosen physical location contains real or dummy data, in order to avoid overwriting the existing useful data block. HIVE proposes a complex 'inverse position map' approach for collision avoidance. Essentially, it maintains a forward position mapping (logical to physical blocks) and a backward/inverse position mapping (physical to logical blocks) for the whole memory. Before each write, both forward and backward mappings are looked up to determine whether the corresponding physical location is vacant or occupied. This approach, however, turns out to be a storage and performance bottleneck because of the size of inverse position map and dual lookups of the position mappings.

A simplistic and obvious solution to this problem would be to use a bit-mask to mark each physical block as vacant or occupied. Based on this intuition, we propose a simplified write-only ORAM scheme called Flat ORAM that adapts the basic construction from HIVE while solving the collisions problem in a much more efficient way. At the core of the algorithm, Flat ORAM introduces a new data structure called Occupancy Map (OccMap): a bit-mask which offers an efficient collision avoidance mechanism. The OccMap records the availability information (occupied/vacant) of every physical location in a highly compressed form (i.e., just 1-bit per cache line). For typical parameter settings, OccMap is about 25× compact compared to HIVE's inverse position map structure. This dense structure allows the OccMap blocks to exploit higher locality, resulting in considerable performance gain. While naively storing the OccMap breaks the ORAM's security, we present how to securely store and manage this structure in a real system. In particular, the paper makes the following contributions:

1. We are the first ones to implement an existing write-only ORAM scheme, HIVE, in the context of secure processors with all state-of-the-art ORAM optimizations, and analyze its performance bottlenecks.
2. A simple write-only ORAM, named Flat ORAM, having an efficient collision avoidance approach is proposed. The micro-architecture of the scheme is discussed in detail and the design space is comprehensively explored. It has also been shown to seamlessly adopt various performance optimizations of its predecessor: Path ORAM.
3. Our simulation results show that, on average, Flat ORAM offers 50% performance gain (up to 75% for DBMS) over HIVE, and only incurs slowdowns of 3× and 1.6× over the insecure DRAM for memory bound Splash2 and SPEC06 benchmarks, respectively.

The rest of the paper is organized as follows: Section 2 describes our adversarial model in detail along with a practical example from the current literature. Section 3 provides the necessary background of fully functional ORAMs and write-only ORAMs. The proposed Flat ORAM scheme along with its security analysis is presented in Section 4, and the detailed construction of its occupancy map structure is shown in Section 5. A few additional optimizations from literature implemented in Flat ORAM are discussed in Section 6. Sections 7 and 8 evaluate Flat ORAM's performance, and we conclude the paper in Section 9.

2. Adversarial Model

We assume a relaxed form of the adversarial model considered in several prior works related to fully functional oblivious RAMs in secure processor settings [26,28,39].

In our model, a user's computation job is outsourced to a cloud, where a trusted processor performs the computation on user's private data. The user's private data is stored (in encrypted form) in the untrusted memory external to the trusted processor, i.e., DRAM. In order to compute on user's private data, the trusted processor interacts with DRAM. In particular, the processor's last level cache (LLC) misses result in reading the data from DRAM, and the evictions from the LLC result in writing the data to DRAM. The cloud service itself is not considered as an adversary, i.e., it does not try to infer any information from the memory access patterns of the user's program. However, since the cloud serves several users at the same time, sharing of critical resources such as DRAM among various applications from different users is inevitable. Among these users being served by the cloud service, we assume a malicious user who is able to monitor remotely (and potentially learn the secret information from) the data write sequences of other users' applications to the DRAM, e.g., by taking frequent snapshots of the victim application's memory via a compromised DMA. Moreover, he may also tamper with the DRAM contents or play replay attacks in order to manipulate other users' applications and/or learn more about their secret data.

To protect the system from such an adversary, we add to the processor chip a 'Write-Only ORAM' controller: an additional trusted hardware module. Now all the off-chip traffic goes to DRAM through the ORAM controller. In order to formally define the security properties satisfied by our ORAM controller, we adapt the write-only ORAM privacy definition from [41] as follows:

Definition 1. ***(Write-Only ORAM Privacy)*** *For every two logical access sequences A_1 and A_2 of infinite length, their corresponding (infinite length) probabilistic access sequences $\mathsf{ORAM}(A_1)$ and $\mathsf{ORAM}(A_2)$ are identically distributed in the following sense: For all positive integers n, if we truncate $\mathsf{ORAM}(A_1)$ and $\mathsf{ORAM}(A_2)$ to their first n accesses, then the truncations $[\mathsf{ORAM}(A_1)]_n$ and $[\mathsf{ORAM}(A_2)]_n$ are identically distributed.*

In other words, memory snapshots only reveal to the adversary the timing of write accesses made to the memory (i.e., leakage over ORAM Timing Channel) instead of their precise access pattern, whereas no trace of any read accesses made to the memory is revealed to the adversary. An important aspect of Definition 1 to note is that it completely isolates the problem of leakage over ORAM Termination Channel from the original problem that ORAM was proposed for, i.e., preventing leakage over memory address channel (which is also the focus of this paper). Notice that the original definition of ORAM [14] does not protect against timing attacks, i.e., it does not obfuscate *when* an access is made to the memory (ORAM Timing Channel) or how long it takes the application to finish (ORAM Termination Channel). The write-only ORAM security definition followed by HIVE [40] also allows leakage over ORAM termination channel, as two memory access sequences generated by the ORAM for two same-length logical access sequences can have different lengths [41]. Therefore, in order to define precise security guarantees offered by our ORAM, we follow Definition 1.

Periodic ORAMs [10] deterministically make ORAM accesses always at regular predefined (publicly known) intervals, thereby preventing leakage over ORAM timing channel and shifting it to the ORAM termination channel. We present a periodic variant of our write-only ORAM to protect leakage over ORAM timing channel. Following the prior works [11,26], (a) we assume that the timing of individual DRAM accesses made during an ORAM access does not reveal any information; (b) we do not protect the leakage over ORAM termination channel (i.e., total number of ORAM or DRAM accesses). The problem of leakage over ORAM termination channel has been addressed in the existing literature [42] where only $\log_2(n)$ bits are leaked for a total of n accesses. A similar approach can easily be applied to the current scheme.

In order to detect malicious tampering of the memory by the adversary, we follow the standard definition of data integrity and freshness [26]:

Definition 2. ***(Write-Only ORAM Integrity)*** *From the processor's perspective, the ORAM behaves like a valid memory with overwhelming probability, and detects any violation to data authenticity and/or freshness.*

Practicality of the Adversarial Model

Modular exponentiation algorithms, such as RSA algorithm, are widely used in public-key cryptography. In general, these algorithms perform computations of the form $y = g^k \mod n$, where the attacker's goal is to find the secret k. Algorithm 1 shows the Montgomery Ladder scheme [30] which performs exponentiation (g^k) through simple square-and-multiply operations. For a given input g and a secret key k, the algorithm performs multiplication and squaring operations on two local variables R_0 and R_1 for each bit of k starting from the most significant bit down to the least significant bit. This algorithm prevents leakage over power side-channel since, regardless of the value of bit k_j, the same number of operations are performed in the same order, hence producing the same power footprint for $k_j = 0$ and $k_j = 1$.

Algorithm 1 Montgomery Ladder

```
Inputs: $g, k = (k_{t-1}, \cdots, k_0)_2$    Output: $y = g^k$
Start:
1: $R_0 \leftarrow 1; R_1 \leftarrow g$
2: for $j = t - 1, 0$ do
3:     if $k_j = 0$ then   $R_1 \leftarrow R_0 R_1$;   $R_0 \leftarrow (R_0)^2$
4:     else   $R_0 \leftarrow R_0 R_1$;   $R_1 \leftarrow (R_1)^2$
5:     end if
6: end for
return $R_0$
```

Notice, however, that the specific order in which R_0 and R_1 are updated in time depends upon the value of k_j. E.g., for $k_j = 0$, R_1 is written first and then R_0 is updated; whereas for $k_j = 1$ the updates are done in the reverse order. This sequence of write access to R_0 and R_1 reveals to the adversary the exact bit values of the secret key k. A recent work [29] demonstrated such an attack where frequent memory snapshots of victim application's data (particularly R_0 and R_1) from the physical memory are taken via a compromised DMA. These snapshots are then correlated in time to determine the sequence of write access to R_0, R_1, which in turn reveals the secret key. The reported time taken by the attack is 3.5 min.

One might argue that under write-back caches, the updates to application's data will only be visible in DRAM once the data is evicted from the last level cache (LLC). This will definitely introduce noise to the precise write-access sequence discussed earlier, hence making the attacker's job difficult. However, he can still collect several 'noisy' sequences of memory snapshots and then run correlation analysis on them to find the secret key k. Furthermore, if the adversary is also a user of the same computer system, he can flush the system caches frequently to reduce the noise in write-access sequence even further.

3. Background of Oblivious RAMs

A full-featured Oblivious RAM [14], or more commonly known as ORAM, is a primitive that obfuscates the user's (i.e., Processor's) access patterns to a storage (i.e., DRAM) such that by monitoring the memory access patterns, an adversary is not able to learn anything about the data being accessed. The ORAM interface transforms the user's access sequence of program addresses into a sequence of ORAM accesses to random looking physical addresses. Since the physical locations being accessed are revealed to the adversary, the ORAM interface guarantees that the physical access pattern is independent of the logical access pattern hence user's potentially data dependent access patterns are not revealed. Furthermore, the data stored in ORAMs should be encrypted using probabilistic encryption to conceal the data content as well as the fact whether or not the content has been updated.

3.1. Path ORAM

Path ORAM [24] is currently the most efficient and well studied ORAM implementation for secure processors. It has two main hardware components: the binary tree storage and the ORAM controller. Binary tree stores the data content of the ORAM and is implemented on DRAM. Each node in the tree can hold up to Z useful data blocks, and any empty slots are filled with dummy blocks. All blocks, real or dummy, are probabilistically encrypted and cannot be distinguished. The path from the root node to the leaf s is defined as path s. ORAM controller is a piece of trusted hardware that controls the tree structure. Besides necessary logic circuits, the ORAM controller contains two main structures, a position map and a stash. The position map is a lookup table that associates the program address a of a data block with a path in the ORAM tree (path s). The stash is a buffer that stores up to a small number of data blocks at a time.

Each data block a in Path ORAM is mapped (randomly) to some path s via the position map, i.e., at any time, the data block a must be stored either on path s, or in the stash. Path ORAM follows the following steps when a request on block a is issued by the processor: (1) The path (leaf) number s of the logical address a is looked up in the position map; (2) All the blocks on path s are read and decrypted, and all real blocks added to the stash; (3) Block a is returned to the processor; (4) The position map of a is updated to a new random leaf s'; (5) As many blocks from stash as possible are encrypted and written on path s, where empty spots are filled with dummy blocks. Step (4) guarantees that when block a is accessed later, a random path will be accessed which is independent of any previously accessed paths. As a result, each ORAM access is random and *unlinkable* regardless of the request pattern.

Path ORAM incurs significant energy and performance penalties compared to insecure DRAM. Under typical settings for secure processors (gigabytes of memory and 64- to 128-byte blocks), Path ORAM has a 20–30 level binary tree where each node typically stores 3 or 4 data blocks [23,39]. This means that each ORAM access reads and writes 60–120 blocks, in contrast to a single read or write operation in an insecure storage system.

3.2. Write-Only ORAMs

In contrast to full-featured ORAMs, a write-only ORAM only obfuscates the patterns of write accesses made to a storage. Write-only ORAM is preferred for performance reasons over fully functional ORAMs in situations where the adversary is not able to monitor the read access patterns.

There has been very limited research work done so far to explore write-only ORAMs, and to best of our knowledge, write-only ORAMs for secure processors have not been explored at all. Li and Datta [43] present a write-only ORAM scheme to be used with Private Information Retrieval (PIR) in order to preserve the privacy of data outsourced to a data center. Although this scheme achieves an amortized write cost of $O(B \log N)$, it incurs a read cost of $O(B.N)$ for a storage of N blocks each of size B. For efficient reads, it requires the client side storage (i.e., the on-chip position map) to be polynomial in N. In a secure processor setting, DRAM reads are usually the major performance bottleneck, and introducing a complexity polynomial in N on this critical path is highly unwanted.

A recent work, HIVE [40], has proposed a write-only ORAM scheme for hidden volumes in hard disk drives. Although HIVE write-only ORAM presented-as-is [40] targets a totally different application, we believe that its parameter settings can be tweaked to be used in the secure processor setting. This paper is the first one to implement HIVE in the secure processor context, and we consider this implementation as the baseline write-only ORAM to be compared with our proposed Flat ORAM.

Roche et al. [44] have proposed a "stash-free" write-only ORAM scheme called Deterministic write-only ORAM (DetWoORAM), and evaluated it for hard disk storage drives. DetWoORAM generates a fixed deterministic physical write access pattern regardless of the logical write access pattern, which effectively means that each write operation is always successful unlike HIVE where a write can occasionally fail and is added to the stash. In other words, the deterministic pattern allows DetWoORAM to be stash free. The cost of being stash-free, however, is additional read/write accesses required to move blocks from a temporary "holding" area to the permanent "main" storage area (cf. [44] for details) which would cause additional performance penalty.

Additionally, DetWoORAM has been implemented in software for the DRAM-Disk boundary. The software implementation allows its complex optimizations to be realized in the system, which we believe will be quite challenging to efficiently implement in hardware. Since in this work, we target the Processor-DRAM boundary for our proposed ORAM which needs to be implemented in hardware, hence our focus is only towards simplified algorithms and optimizations which can easily be synthesized in hardware without substantial area overhead. Therefore, in this paper, we do not implement the DetWoORAM for Processor-DRAM boundary and experimentally evaluate it. Instead, we conduct an analytical performance evaluation comparing it with our proposed Flat ORAM scheme in Section 7.

4. Flat ORAM Scheme

At a high level, the proposed Flat ORAM scheme adapts the basics from HIVE's [40] construction while improving on its data collision avoidance solution by introducing an efficient bit-mask based approach, which is a key factor resulting in substantial performance gain. Furthermore, Flat ORAM inherits several architectural optimizations from Path ORAM benefiting from years of research efforts in this domain.

In this section, we first present the core algorithm of Flat ORAM, then we discuss its architectural details and various optimizations for a practical implementation.

4.1. Fundamental Data Structures

4.1.1. Data Array

Flat ORAM replaces the binary tree structure of Path ORAM with a flat or linear array of data blocks; hence termed as 'Flat' ORAM. Since read operations are not visible to the adversary in our model, one does not need to hide the read location as done in Path ORAM by accessing all the locations residing on the path of desired data block. This allows Flat ORAM to avoid the tree structure of Path ORAM altogether and have the data stored in a linear array where only the desired data block is read.

4.1.2. Position Map (PosMap)

PosMap structure maintains randomized mappings of logical blocks to physical locations. Flat ORAM adopts the PosMap structure from a standard Path ORAM. However there is one subtle difference between PosMap of Path ORAM and Flat ORAM. In Path ORAM, PosMap stores a path number for each logical block and the block can reside anywhere on that path. In contrast, PosMap in Flat ORAM stores the exact physical address where a logical block is stored in the linear data array.

4.1.3. Occupancy Map (OccMap)

OccMap is a newly introduced structure in Flat ORAM. It is essentially a large bit-mask where each bit corresponds to a physical location (i.e., a cache line). The binary value of each bit represents whether the corresponding physical block contains fresh/real or outdated/dummy data. OccMap is of crucial importance to avoid data collisions, and hence for the correctness of Flat ORAM. A collision happens when a physical location, which is randomly chosen to store a logical block, already contains useful data which cannot be overwritten. Managing the OccMap securely and efficiently is a major challenge which we address in Section 5 in detail.

4.1.4. Stash

Stash, also adapted from Path ORAM, is a small buffer in the trusted memory to temporarily hold data blocks evicted from the processor's last level cache (LLC). A slight but crucial modification, however, is that Flat ORAM only buffers dirty data blocks (blocks with modified data) in the stash; while clean (or unmodified) blocks evicted from the LLC are simply ignored since a valid copy of these blocks already exists in the main memory. This modification is significantly beneficial for performance.

4.2. Basic Algorithm

Let N be the total number of logical data blocks that we want to securely store in our ORAM, which is implemented on top of a DRAM; and let each data block be of size B bytes. Let P be the number of physical blocks that our DRAM can physically store, i.e., the DRAM capacity (where $P \geq N$).

Initial Setup: Algorithm 2 shows the setup phase of our scheme. Two null-initialized arrays PosMap and OccMap, corresponding to position and occupancy map of size N and P entries respectively, are allocated. For now, we assume that both PosMap and OccMap reside on-chip in the trusted memory to which the adversary has no access. However, since these arrays can be quite large and the trusted memory is quite constrained, we later on show how this problem is solved. Initially, since all physical blocks are empty, each OccMap entry is set to 0. Now, each logical block is mapped to a uniformly random physical block, i.e., PosMap is initialized, while avoiding any collisions using OccMap. The OccMap is updated along the way in order to mark those physical locations which have been assigned to a logical block as 'occupied'. Notice that in order to minimize the probability of collision, P should be sufficiently larger than N, e.g., $P \approx 2N$ gives a 50% collision probability.

Algorithm 2 Flat ORAM Initialization.

```
1: procedure INITIALIZE( $N, B, P$ )
2:     PosMap := $\{\perp\}^N$                          ▷ Empty Position Map.
3:     OccMap := $\{0\}^P$                              ▷ Empty Occupancy Map.
4:     for $j \in \{1, \cdots, N\}$ do
5:         loop
6:             $r \leftarrow$ UNIFORMRAND($1, \cdots, P$)
7:             if OccMap[$r$] == 0 then                 ▷ If vacant
8:                 OccMap[$r$] := 1                     ▷ Mark Occupied.
9:                 PosMap[$j$] := $r$                   ▷ Record position.
10:                break
11:            end if
12:        end loop
13:    end for
14: end procedure
```

Reads: The procedures to read/write a data block corresponding to the virtual address a from/to the ORAM are shown in Algorithm 3. A read operation is straightforward as it does not need to be obfuscated. The PosMap entry for the logical block a is looked up, and the encrypted data is read through normal DRAM read. The data is decrypted and returned to the LLC along with its current physical position s. The location s is stored in the tag array of the LLC, and proves to be useful upon eviction of the data from the LLC.

Writes: Since write operations should be non-blocking in a secure processor setting, the ORAM writes are performed in two steps. Whenever a data block is evicted from the LLC, it is first simply added to the stash queue, without incurring any latency. While the processor moves on to computing on other data in its registers/caches, the ORAM controller then works in the background to evict the block from stash to the DRAM. A block a to be written is picked from the stash, and a new uniformly random physical position $s_{\texttt{new}}$ is chosen for this block. The OccMap is looked up to determine whether the location $s_{\texttt{new}}$ is vacant. If so, the write operation proceeds by simply recording the new position $s_{\texttt{new}}$ for block a in PosMap, updating the OccMap entries for $s_{\texttt{new}}$ and $s_{\texttt{old}}$ accordingly, and finally writing encrypted data at location $s_{\texttt{new}}$. Otherwise if the location $s_{\texttt{new}}$ is already occupied by some useful data block, the probability of which is N/P, the existing data block is read, decrypted, re-encrypted under probabilistic encryption and written back. A new random position is then chosen for the block a and the above mentioned process is repeated until a vacant location is found to evict the block. The encryption/decryption algorithms $\mathsf{Enc}_K/\mathsf{Dec}_K$ implement probabilistic encryption, e.g., AES counter mode, as done in prior works [26,39]. Notice that storing $s_{\texttt{old}}$ along with the data upon reads will save extra ORAM accesses to lookup $s_{\texttt{old}}$ from the recursive PosMap (cf. Section 4.5).

An implementation detail related to read operations is that upon a LLC miss, in some cases the data block might be supplied by the stash instead of the DRAM. Since the stash serves as another level of cache hierarchy for the data blocks, it might contain fresh data recently evicted from the LLC that is yet to be written back to DRAM. In such a scenario, the data block is refilled to LLC by the stash which contains the freshest data, which is also beneficial for performance.

Algorithm 3 Basic Flat ORAM considering all PosMap and OccMap is on-chip. Following procedures show reading, writing and eviction of a logical block a from the ORAM.

```
1: procedure ORAMREAD($a$)
2:     $s := \mathsf{PosMap}[a]$                                  ▷ Lookup position
3:     $data := \mathsf{Dec_K}(\text{DRAMREAD}(s))$
4:     return $(s, data)$                                         ▷ Position is also returned.
5: end procedure

1: procedure ORAMWRITE($a, s_{\texttt{old}}, data$)
2:     $Stash := Stash \cup \{(a, s_{\texttt{old}}, data)\}$      ▷ Add to Stash
3:     return
4: end procedure

1: procedure EVICTSTASH
2:     $(a, s_{\texttt{old}}, data) \leftarrow Stash$               ▷ Read from Stash
3:     loop
4:         $s_{\texttt{new}} \leftarrow \text{UNIFORMRAND}(1, \cdots, P)$
5:         if $\mathsf{OccMap}[s_{\texttt{new}}] == 0$ then          ▷ If vacant
6:             $\mathsf{OccMap}[s_{\texttt{new}}] := 1$              ▷ Mark as Occupied.
7:             $\mathsf{OccMap}[s_{\texttt{old}}] := 0$              ▷ Vacate old block.
8:             $\mathsf{PosMap}[a] := s_{\texttt{new}}$              ▷ Record position.
9:             DRAMWRITE($s_{\texttt{new}}, \mathsf{Enc_K}(data)$)
10:            $Stash := Stash \setminus \{(a, s_{\texttt{old}}, data)\}$
11:            break
12:        else                                                  ▷ If occupied.
13:            $data' := \mathsf{Dec_K}(\text{DRAMREAD}(s_{\texttt{new}}))$
14:            DRAMWRITE($s_{\texttt{new}}, \mathsf{Enc_K}(data')$)
15:        end if
16:    end loop
17: end procedure
```

4.3. Avoiding Redundant Memory Accesses

The fact that the adversary cannot see read accesses allows Flat ORAM to avoid almost all the redundancy incurred by a fully functional ORAM (e.g., Path ORAM). Instead of reading/writing a whole path for each read/write access as done in Path ORAM, Flat ORAM simply reads/writes only the desired block directly given its physical location from the PosMap. This is fundamentally where the write-only ORAMs (i.e., HIVE [40], Flat ORAM) get the performance edge over the full-featured ORAMs. However, the question arises whether Flat ORAM is still secure after eliminating the redundant accesses.

4.4. Security

Privacy: Consider any two logical write-access sequences O_0 and O_1 of the same length. In EVICTSTASH procedure (cf. Algorithm 3), a physical block chosen uniformly at random out of P blocks is always modified regardless of it being vacant or occupied. Therefore, the write accesses generated by Flat ORAM while executing either of the two logical access sequences O_0 and O_1 will update memory locations uniformly at random throughout the whole physical memory space. As a result, an adversary monitoring these updates cannot distinguish between real vs. dummy blocks, and in turn the two sequences O_0 and O_1 seem computationally indistinguishable.

Furthermore, notice that in Path ORAM the purpose of accessing a whole path instead of just one block upon each read and write access is to prevent linkability between a write and a following read access to the same logical block. In Flat ORAM's model, however, since the adversary cannot see the read accesses at all, therefore the linkability problem would never arise as long as each logical data block is written to a new random location every time it is evicted (which is guaranteed by Flat ORAM algorithm). Although HIVE proposes a constant k-factor redundancy upon each data write, we argue that it is unnecessary for the desired security as explained above, and can be avoided to gain performance. Hence, the basic algorithm of Flat ORAM presented above guarantees the desired privacy property of our write-only ORAM (cf. Definition 1).

Integrity: Next, we move on to making the basic Flat ORAM practical for a real system. The main challenge is to get rid of the huge on-chip memory requirements imposed by PosMap and OccMap. While addressing the PosMap management problem, we discuss an existing efficient memory integrity verification technique from Path ORAM domain called *PMMAC* [26] (cf. Section 6.2) which satisfies our integrity definition (cf. Definition 2).

Stash Management: Another critical missing piece is to prevent the unlikely event of stash overflow for a small constant sized stash, as such an event could break the privacy guarantees offered by Flat ORAM. We completely eliminate the possibility of a stash overflow event by using a proven technique called *Background Eviction* [39]. We present a detailed discussion about the stash size under Background Eviction technique in Section 5.4.

4.5. Recursive Position Map & Position map Lookaside Buffer (PLB)

In order to squeeze the on-chip PosMap size, a standard recursive construction of position map [22] is used. In a 2-level recursive position map, for instance, the original PosMap structure is stored in another set of data blocks which we call a hierarchy of position map, and the PosMap of the first hierarchy is stored in the trusted memory on-chip (Figure 1). The above trick can be repeated, i.e., adding more hierarchies of position map to further reduce the final position map size at the expense of increased latency. Notice that all the position map hierarchies (except for the final position map) are stored in the untrusted DRAM along with the actual data blocks, and can be treated as regular data blocks; this technique is called Unified ORAM [26].

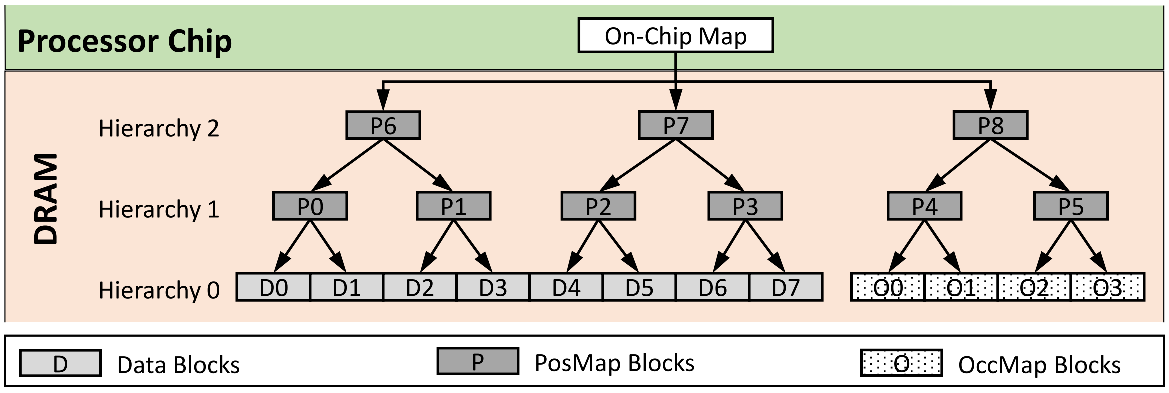


Figure 1. Logical view of Occupancy Map (OccMap) organization.

Unified ORAM scheme reduces the performance penalty of recursion by caching position map ORAM blocks in a Position map Lookaside Buffer (PLB) to exploit locality (similar to the TLB exploiting locality in page tables). To hide whether a position map access hits or misses in the cache, Unified ORAM stores both data and position map blocks in the same binary tree. Further compression of PosMap structure is done by using Compressed Position Map technique discussed in Section 6.1.

4.6. Background Eviction

Stash (cf. Section 4.1.4) is a small buffer to temporarily hold the dirty data blocks evicted from the LLC/PLB. If at any time, the rate of blocks being added to the stash becomes higher than the rate of evictions from the stash, the blocks may accumulate in stash causing a stash overflow. Background eviction [39] is a proven and secure technique proposed for Path ORAM to prevent stash overflow.

The key idea of background evictions is to temporarily stop serving the real requests (which increase stash occupancy) and issue background evictions or so-called dummy accesses (which decrease stash occupancy) when the stash is full.

We use background eviction technique to eliminate the possibility of a stash overflow event. When the stash is full, the ORAM controller suspends the read requests which consequently stops any write-back requests preventing the stash occupancy to increase further. Then it simply chooses random locations and, if vacant, evicts the blocks from the stash until the stash occupancy is reduced to a safe threshold. The probability of a successful eviction in each attempt is determined by the DRAM utilization parameter, i.e., the ratio of occupied blocks to the total blocks in DRAM. In our experiments, we choose a utilization of $\approx 1/2$, therefore each eviction attempt has $\approx 50\%$ probability of success. Note that background evictions essentially push the problem of stash overflow to the program's termination channel. Configuring the DRAM utilization to be less than 1 guarantees the termination, and we demonstrate good performance for a utilization of $1/2$ in Section 8. Although it is true that background eviction may have different effect on the total runtime for different applications, or even for different inputs for the same application which leak the information about data locality etc. However, the same argument applies to Path ORAM based systems, and also for other system components as well, such as enabling vs. disabling branch prediction or the L3 cache, etc. Protecting any leakage through the program's termination time is out of scope of this paper (cf. Section 2).

4.7. Periodic ORAM

As mentioned in our adversarial model, the core definition of ORAM [14] does not address leakage over ORAM timing or termination channel (cf. Section 2). Likewise, the fundamental algorithm of Flat ORAM (Algorithm 3) does not target to prevent these leakages. Therefore in order to protect the ORAM timing channel, we adapt the Flat ORAM algorithm to issue periodic ORAM accesses, while maintaining its security guarantees.

In the literature, periodic variants of Path ORAM have been presented [10] which simply always issue ORAM requests at regular periodic intervals. However, under write-only ORAMs, such a straightforward periodic approach would break the security as explained below. Since in write-only ORAMs, the read requests do not leave a trace, therefore for a logical access sequence of (`Write`, `Read`, `Write`), the adversary will only see two writes occurring at times 0 and $2T$ for T being the interval between two ORAM accesses. The access at time T will be omitted which reveals to the adversary that a read request was made at this time.

To fix this problem, we modify the Flat ORAM algorithm as follows. Among the periodic access, for every real read request to physical block s, another randomly chosen physical block s' is also read. Block s is consumed by the processor, whereas block s' is re-encrypted (under probabilistic encryption) and written back to the same location from where it was read. This would always result in update(s) to the memory after each and every time period T.

Security: A few things should be noted: First, it does not matter whether the location s' contains real or dummy data, because the plain-text data content is never modified but just re-encrypted. Second, writing back s' is indistinguishable from a real write request as this location is chosen uniformly at random. Third, this write to s' does not reveal any trace of the actual read of s as the two locations are totally independent.

We present our simulation results for periodic Flat ORAM in the evaluation section.

5. Efficient Collision Avoidance

In Sections 4.3 and 4.4, we discuss how Flat ORAM outperforms Path ORAM by avoiding redundant memory accesses. However, an immediate consequence of this is the problem of collisions which now becomes the main performance bottleneck. A collision refers to a scenario when a physical location s, which is randomly chosen to write a logical block a, already contains useful data which cannot be overwritten. The overall efficiency of such a write-only ORAM scheme boils down to its collision avoidance mechanism. In the following subsections, we discuss the occupancy map based collision avoidance mechanism of Flat ORAM in detail and compare it with HIVE's 'inverse position map' based collision avoidance scheme.

5.1. Inverse Position Map Approach

Since a read access must not leave its trace in the memory in order to avoid the linkability problem, a naive approach of marking the physical location as 'vacant' by writing 'dummy' data to it upon each read is not possible. HIVE [40] proposes an inverse position map structure that maps each physical location to a logical address. Before each write operation to a physical location s, a potential collision check is performed which involves two steps. First the logical address a linked to s is looked up via the inverse position map. Then the regular position map is looked up to find out the most recent physical location s' linked to the logical address a. If $s = s'$ then since the two mappings are synchronized, it shows that s contains useful data, hence a collision has occurred. Otherwise, if $s \neq s'$ then this means that the entry for s in the inverse position map is outdated, and block a has now moved to a new location s'. Therefore, the current location can be overwritten, hence no collision.

HIVE stores the encrypted inverse position map structure in the untrusted storage at a fixed location. With each physical block being updated, the corresponding inverse position map entry is also updated. Since write-only ORAMs do not hide the physical block ID of the updated block, therefore revealing the position of the corresponding inverse position map entry does not leak any secret information.

We demonstrate in our evaluations that the large size of inverse position map approach introduces storage as well as performance overheads. For a system with a block size of B bytes and total N logical blocks, inverse position map requires $\log_2(N)$ additional bits space for each of the P physical blocks. Crucially, this large size of a single inverse position map entry restricts the total number of entries per block to a small constant, which leads to less locality within a block. This results in performance degradation.

5.2. Occupancy Map Approach

In the simplified OccMap based approach, each of the P physical blocks requires just one additional bit to store the occupancy information (vacant/occupied). In terms of storage, this gives $\log_2(N)$ times improvement over HIVE.

5.2.1. Insecurely Managing the Occupancy Map

The OccMap array bits are first sliced into chunks equal to the ORAM block size (B bytes). We call these chunks the OccMap Blocks. Notice that each OccMap block contains occupancy information of $8B$ physical locations (i.e., 8 bits per byte; 1-bit per location). Now the challenge is to efficiently store these blocks somewhere off-chip. A naive approach would be to encrypt OccMap blocks under probabilistic encryption, and store them contiguously in a dedicated fixed area in DRAM. However under Flat ORAM algorithm, this approach would lead to a serious security flaw which is explained below.

If the OccMap blocks are stored contiguously at a fixed location in DRAM, an adversary can easily identify the corresponding OccMap block for a given a physical address; and for a given OccMap block, he can identify the contiguous range of corresponding $8B$ physical locations. With that in mind, when a data block a (previously read from $s_{\texttt{old}}$) is evicted from the stash and written to location $s_{\texttt{new}}$ (cf. Algorithm 3), the old OccMap entry is marked as 'vacant' and the new OccMap entry is marked as 'occupied'; i.e., two OccMap blocks $O_{\texttt{old}}$ and $O_{\texttt{new}}$ are updated. Furthermore, the $s_{\texttt{new}}$ location, which falls in the contiguous range covered by one of the two updated OccMap blocks, is also updated with the actual data—thereby revealing the identity of $O_{\texttt{new}}$. This reveals to the adversary that a logical data block was previously read from some location within the small contiguous range covered by $O_{\texttt{old}}$, and it is now written to location $s_{\texttt{new}}$ (i.e., coarse grained linkability). Recording several such instances of read-write access pairs and linking them together in a chain reveals the precise pattern of movement of logical block *a* across the whole memory.

5.2.2. Securely Managing the Occupancy Map

To avoid this problem, we treat the OccMap blocks as regular data blocks, i.e., OccMap blocks are encrypted and also randomly distributed throughout the whole DRAM, and tracked by the regular PosMap. Figure 1 shows the logical organization of data, PosMap and OccMap blocks. The OccMap blocks are added as 'additional' data blocks at the data hierarchy (Hierarchy 0). Then the recursive position map hierarchies are constructed on top of the complete data set (data blocks and OccMap blocks). Every time an OccMap block is updated, it is mapped to a new random location and hence avoids the linkability problem. We realize that this approach results in overall more position map blocks, however for practical parameters settings (e.g., 128 B block size, 8 GB DRAM, 4 GB working set), it does not result in an additional PosMap hierarchy. Therefore the recursive position map lookup latency is unaffected.

5.3. Performance Related Optimizations

Now that we have discussed how to securely store the occupancy map, we move on to discuss some performance related optimizations implemented in Flat ORAM.

5.3.1. Locality in OccMap Blocks

For realistic parameters, e.g., 128 bytes block size, each OccMap block contains occupancy information of 1024 physical locations. This factor is termed as OccMap scaling factor. The dense structure of OccMap blocks offers an opportunity to exploit spatial locality within a block. In other words, for a large scaling factor it is more likely that two randomly chosen physical locations will be covered by the same OccMap block, as compared to a small scaling factor. In order to also benefit from such locality, we cache the OccMap blocks as well in PLB along with the PosMap blocks. For a fair comparison with HIVE in our experiments, we also model the caching of HIVE's inverse position map blocks in PLB. Our experiments confirm that the OccMap blocks show a higher PLB hit rate as compared to HIVE's inverse position map blocks cached in PLB in the same manner. The reason is that, for the same parameters, the scaling factor of inverse position map approach is just about 40 which results in a larger size of the data structure and hence more capacity-misses from the PLB.

5.3.2. Dirty vs. Clean Evictions from LLC & PLB

An eviction of a block from the LLC where the data content of the block has not been modified is called a clean eviction, whereas an eviction where the data has been modified is called dirty eviction. In Path ORAM, since all the read operations also need to be obfuscated, therefore following a read operation, when a block gets evicted from the LLC, it must be re-written to a new random location even if its data is unmodified, i.e., a clean eviction. This is crucial for Path ORAM's security as it guarantees that successive reads to the same logical block result in random paths being accessed. This notion is termed as *read-erase*, which assumes that the data will be erased from the memory once it is read.

In write-only ORAMs, however, since the read access patterns are not revealed therefore the notion of read-erase is not necessary. A data block can be read from the same location as often as needed as long as it's contents are not modified. We implement this relaxed model in Flat ORAM which greatly improves performance. Essentially, upon a clean eviction from the LLC, the block can simply be discarded since one useful copy of the data is still stored in the DRAM. Same reasoning applies to the clean evictions from the PLB. Only the dirty evictions are added to the stash to be written back at a random location in the memory.

5.4. Implications on PLB & Stash Size

Each dirty eviction requires not only the corresponding data block to be updated but also the two related OccMap blocks which store the new and old occupancy information. In order to relocate these blocks to new random positions, the 'd' hierarchies of corresponding PosMap blocks will need to be updated and this in turn implies updating their related OccMap blocks, and so on. If not prevented, this avalanche effect will repeatedly fill the stash implying background evictions which stop serving real requests and increase the termination time. For a large enough PLB with respect to a benchmark's locality, most of the required OccMap blocks during the benchmark's execution will be in the PLB. This prevents the avalanche effect most of the time (as our evaluation shows) since the OccMap blocks in PLB can be directly updated.

Even if all necessary OccMap blocks are in PLB, a dirty eviction still requires the data block with its d PosMap blocks to be updated. Each of these $d+1$ blocks is successfully evicted from the stash with probability $1/2$, determined by the DRAM utilization, on each attempt. The probability that exactly m attempts are needed to evict all $d+1$ blocks is equal to $\binom{m-1}{d}/2^m$. This probability becomes very small for m equal to a small constant c times $d \log d$. If the dirty eviction rate (per DRAM access) is at most $1/c$, then the stash size will mostly be contained to a small size (e.g., 100 blocks for $d = 4$) so that additional background eviction which stops serving real requests is not needed.

Notice that the presented write-only ORAM is not asymptotically efficient: In order to show at most a constant factor increase in termination time with overwhelming probability, a proper argument needs to show a small probability of requiring background eviction which halts normal execution. An argument based on M/D/1 queuing theory or 1-D random walks needs the OccMap to be always within the PLB and this means that the effective stash size as compared to Path ORAM's stash definition includes this PLB which scales linearly with N and is not $O(logN)$.

6. Adopting More Existing Tricks

Here we discuss a few more architectural optimizations from the Path ORAM paradigm which can be flawlessly incorporated and are implemented in Flat ORAM for further improvements and features.

6.1. Compressed Position Map

The recursive position map for a total of N logical data blocks creates $\lceil \log_b(N) \rceil$ hierarchies of position map. Here b represents the number of positions stored in one PosMap block, and is called PosMap scale factor. A higher value of PosMap scale factor would result in less number of PosMap hierarchies and hence yield better performance.

To achieve this goal, Compressed Position Map [26] has been proposed, which results in less PosMap hierarchies than uncompressed PosMap. The basic idea is to store a monotonically increasing counter in the PosMap entry for each logical data block. This counter along with the block's logical address is used as a 'seed' to a keyed pseudo-random function in order to compute a random position for the block. Every time a block is to be written, its PosMap counter is first incremented so that a new random position is generated by the pseudo-random function for the block. To compress these counters to a feasible size, [26] presents an optimization using a big group counter and several small individual counters per PosMap block. We refer the readers who might be interested in more details to the above citation.

We tweak the compressed PosMap technique for Flat ORAM. The key modification is that the counter for any block to be evicted is incremented even upon unsuccessful eviction attempts, i.e., even if a collision is detected. It is important because otherwise the pseudo-random function will generate the same random location over and over which is already occupied, and hence the block will never be evicted.

6.2. Integrity Verification (PMMAC)

Flat ORAM also implements an efficient memory integrity verification technique termed as PosMap MAC (PMMAC) [26]. MAC refers to Message Authentication Code, i.e., a keyed cryptographic hash used to verify the authenticity of a message/data. PMMAC scheme leverages the per-block counters of compressed PosMap to perform MAC checks on the data upon reads.

Suppose a logical block a has a counter c, then upon writes, the ORAM controller computes a MAC $h = \mathsf{MAC}_K(a \,||\, c \,||\, data)$ using the secret key K and writes the tuple $(h, data)$ to the DRAM. Upon reads, the potentially tampered data tuple $(h^*, data^*)$ is read. The ORAM controller recomputes $h = \mathsf{MAC}_K(a \,||\, c \,||\, data^*)$ and checks whether $h = h^*$. If so, the data integrity is verified. Also, since the counter is incremented upon every write, the freshness of the data is also verified, i.e., integrity check guarantees that the most recently written data has been read.

7. Analytical Evaluation

In this section, we conduct an analytical evaluation of DetWoORAM [44] and analyze the performance implications of its algorithmic details to compare with Flat ORAM.

7.1. Comparison with DetWoORAM

At a high level the DetWoORAM algorithm seems simple with its deterministic physical access pattern and stash-free design. However, a closer look at its implementation details reveals its complexity and performance implications compared to Flat ORAM. Some of those details are highlighted below in contrast to Flat ORAM showing that Flat ORAM would result in better performance with its simpler implementation compared to DetWoORAM under similar configuration settings.

7.1.1. Read Latency

For CPU performance, the load miss latency or load-to-use latency is critical. Flat ORAM minimizes the read latency as much as possible—since it lies on the critical path for CPU performance—by having minimal number of DRAM reads in a write-only ORAM read operation, i.e., (1) lookup the position map, and (2) read the data block.

DetWoORAM on the other hand incurs additional read operations during a write-only ORAM read which is essentially a cost of a truly stash-free write-only ORAM. Each DetWoORAM read operation needs to (1) lookup the position map; (2) read a data block from the "main" area; (3) perform the freshness check; and (4) finally read the fresh block from holding area if freshness check failed.

Notice that Flat ORAM only needs to do steps (1) and (2), whereas DetWoORAM would incur additional latency to a write-only read operation by having extra steps (3) and (4), hurting performance.

7.1.2. Write Complexity

Flat ORAM write operation has following steps: (1) lookup the occupancy map; (2) write the data block to new randomly chosen position; and (3) update the position map. For a 50% DRAM utilization ratio, the Flat ORAM write operation succeeds with about 50% probability.

On the other hand, DetWoORAM write operation always succeeds, but involves much more read/write operations compared to Flat ORAM. DetWoORAM write operation has following steps: (1) write fresh data block to holding area; (2) read stale block from main area for computing freshness metadata; and (3) update the position map with new position info and freshness metadata; (4) refresh a main area block by (4a) lookup its position map, (4b) read it from the main area, (4c) perform freshness check, (4d) read the fresh block from holding area if freshness check failed, and finally (4e) write the fresh block to main area to complete refresh operation.

Notice that steps (1) to (3) are somewhat equivalent in both write-only ORAM techniques, and involve a read, a write and a position map update. However, DetWoORAM incurs extra reads/writes to perform the refresh operation, i.e., steps (4a)–(4e) that Flat ORAM does not need to do, hence likely resulting in better performance.

7.1.3. Occupancy Map vs. Freshness Check with One-Bit Diff Technique

In this subsection, we compare the metadata bits required per block for freshness check by DetWoORAM and for collision avoidance by Flat ORAM.

DetWoORAM uses much more bits per block with its 'position map pointers' approach than the occupancy map (OccMap) structure proposed by Flat ORAM. For each data block, DetWoORAM's position map pointer of the form (a_h, o, q) requires $\log_2(B)$ bits for the offset $o \in [0, B)$, and one bit for $q \in \{0, 1\}$, where B denotes the block size in bits, and q indicates the bit value of fresh data at offset o in the block (cf. [44] for details). Hence a total of $(\log_2(B) + 1)$ bits are required per block for freshness check metadata.

On the other hand, Flat ORAM only requires one bit per block to achieve collision avoidance. Furthermore, we store these bits in separate blocks (OccMap) and not within position map blocks unlike DetWoORAM. The compact occupancy map structure and dedicated OccMap blocks not only save space, but also allow us to exploit spacial locality in occupancy map, as shown in the Section 8, since each OccMap block contains occupancy metadata for hundreds of data blocks.

7.1.4. Position Map Management

Flat ORAM uses a recursive position map technique coupled with a position map cache PLB. The recursive structure allows to exploit the spatial locality in the recursion tree of the position map (cf. Figure 1) as the top part of the tree is mostly cached in the PLB. As a result, the average number of physical memory accesses made by Flat ORAM per logical memory access does not see an exponential blow-up, as seen by experimental performance evaluation in Section 8. Overall, FlatORAM performance overhead on average stays under $2\times$ on Splash2 and SPEC06 workloads, and $2.5\times$–$4\times$ on DBMS workloads. Whereas, DetWoORAM for random access patterns on Solid State Disks (the closest comparison with DRAM based system like Flat ORAM) shows about $4.5\times$ overhead.

7.1.5. Software vs. Hardware Implementation

FlatORAM algorithm and optimizations are designed for secure processor environment with a hardware implementation in mind, which greatly limits the algorithm complexity. On the other hand, DetWoORAM is designed for storage disks and implemented in software which allows greater flexibility in terms of algorithmic operations, e.g., loops, iterators etc. We believe it would be highly challenging to efficiently implement DetWoORAM in hardware with all its algorithmic details.

8. Experimental Evaluation

8.1. Methodology

We use Graphite [45] to model different ORAM schemes in all our experiments. Graphite simulates a tiled multi-core chip. The hardware configurations are listed in Table 1. We assume there is only one memory controller on the chip, and all ORAM accesses are serialized. The DRAM in Graphite is simply modeled by a flat latency. The 16 GB/s is calculated assuming a 1 GHz chip with 128 pins, and pins are the bottleneck of the data transfer.

We use Splash-2 [46], SPEC06 [47], and two OLTP database management system (DBMS) [48] workloads, namely YCSB [49] and TPCC [50], to evaluate our Flat ORAM scheme (flat_oram) against various baselines. Three baseline designs are used for comparison: the insecure baseline using normal DRAM (dram), the state of the art Path ORAM with dynamic prefetching [28] (path_oram), and an adaptation of the write-only ORAM scheme from HIVE (hive) in the context of secure processor architectures with several additional optimizations. For all ORAM schemes, we enable the PLB, the compressed position map, and integrity verification. The default parameters for ORAM schemes are shown in Table 1. Unless otherwise stated, all the experiments use these ORAM parameters.

Table 1. System configuration.

Secure Processor Configuration	
Core model	1 GHz, in order core
Total Cores	4
L1 I/D Cache	32 KB, 4-way
Shared L2 cache	512 KB per tile, 8-way
Cacheline (block) size	128 Bytes
DRAM bandwidth	16 GB/s
Conventional DRAM latency	100 cycles
Default ORAM Configuration	
ORAM Capacity	8 GB
Working Set Size	4 GB
Number of ORAM hierarchies	4
ORAM Block size	128 Bytes
PLB Size	32 KB
Stash Size	100 Blocks
Compressed PosMap & Integrity	Enabled

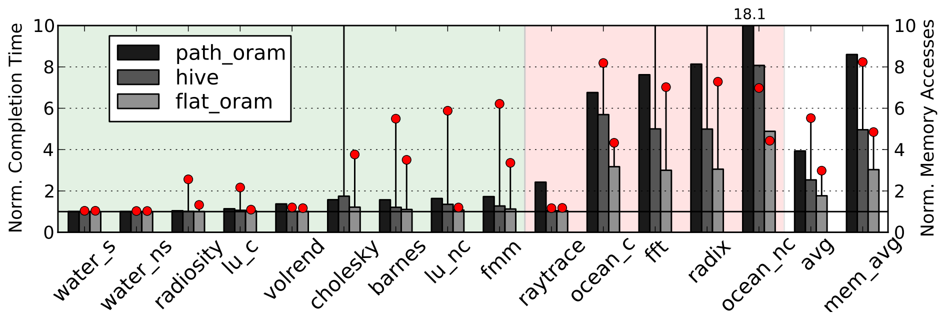
8.2. Performance Comparison

Although all ORAM schemes incur performance slowdown over DRAM, however it is important to note that this slowdown is proportional to the memory intensiveness of the application. Memory-bound applications suffer from higher performance degradation than compute bound applications. Figure 2a–c show normalized completion times (shown by solid bars) of Splash2, SPEC06, and DBMS benchmarks with respect to DRAM. Splash2 and SPEC06 benchmarks are sorted in ascending order of slowdowns over DRAM from left to right. We consider all the benchmarks with less than 2× overhead as 'Computation Intensive' benchmarks (plotted over green background) and all those with more than 2× overhead as 'Memory Intensive' benchmarks (plotted over red background).

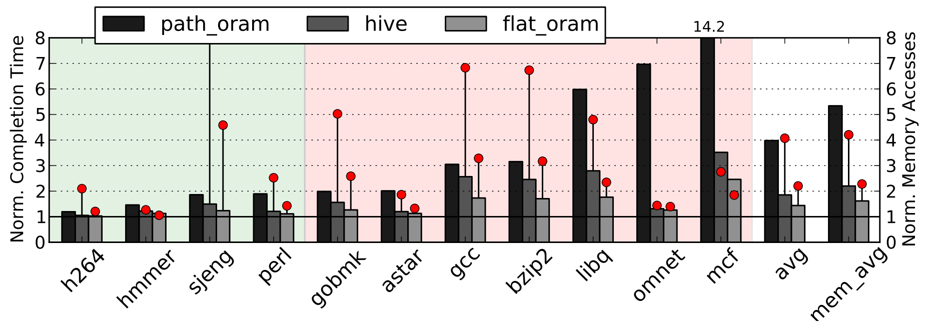
Clearly, Path ORAM incurs the highest overhead, as expected, among all three ORAM schemes because it is a fully functional ORAM which provides higher security. However, the point of presenting this comparison is to convince the readers that using Path ORAM for only write-access protection is indeed an overkill when better alternatives (e.g., HIVE, Flat ORAM) exist. On average, Path ORAM incurs about 8.6× slowdown for Splash2 and 5.3× for SPEC06 memory intensive benchmarks (`mem_avg`). TPCC and YCSB incur 7.2× and 9.3× slowdowns, respectively.

HIVE also shows significant performance degradation compared to Flat ORAM for memory intensive benchmarks (`ocean_contiguous`, `ocean_non_contiguous`, `mcf`, `tpcc`, `ycsb`). The average slowdown of HIVE adaptation for memory bound Splash2 and SPEC06 workloads even after several additional optimizations is 5× and 2.2× respectively. Whereas Flat ORAM outperforms HIVE by up to 50% performance gain on average, having respective average slowdowns of 2× and 1.6×. For DBMS, the performance gain of Flat ORAM over HIVE approach up to 75%.

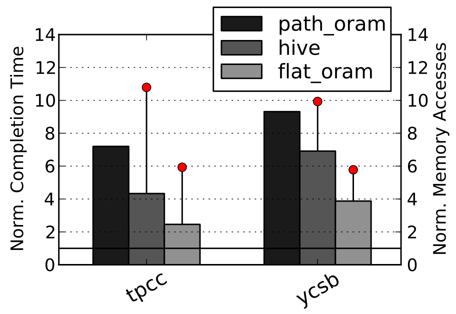
This performance gap is primarily because the inverse position map approach of HIVE results in significantly increased number of additional DRAM accesses. Figure 2 also shows normalized total number of DRAM accesses w.r.t. insecure DRAM system (shown by red markers) for HIVE and Flat ORAM. These numbers include both the DRAM accesses issued to serve regular ORAM requests and also the ones caused by background evictions (cf. Section 4.6). The normalized access count for Path ORAM is around 200 on average, and is not shown on the plots. It can be seen that HIVE issues 8.2 and 4.2 DRAM accesses as opposed to Flat ORAM's 4.8 and 2.3 accesses on average for each request issued by the processor for memory intensive Splash2 and SPEC06 workloads respectively.



(**a**) Splash2



(**b**) SPEC06

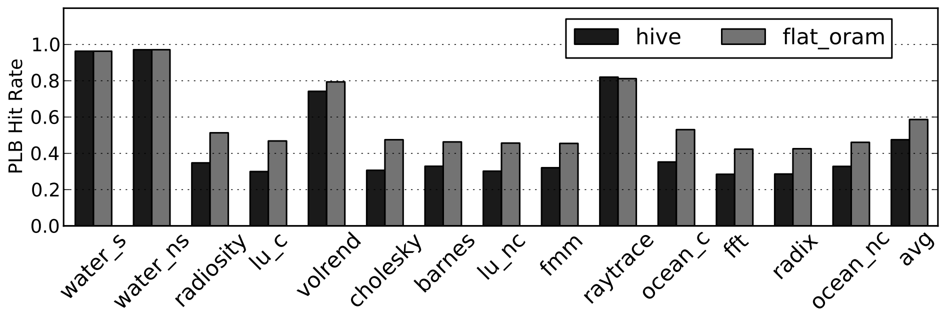


(**c**) DBMS

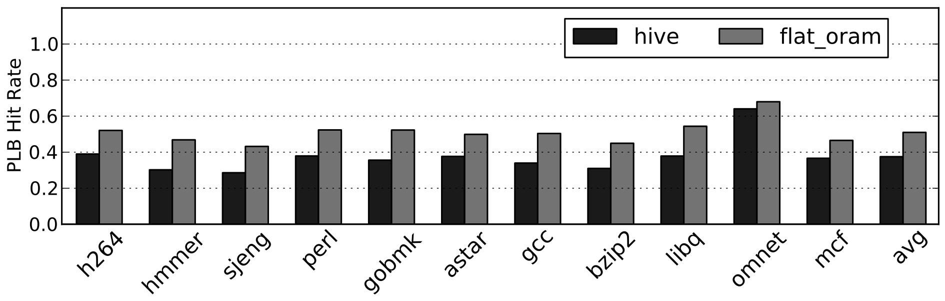
Figure 2. Normalized completion time and memory accesses with respect to insecure DRAM.

The reason for higher number of DRAM accesses from HIVE can be found in Figure 3, which shows the overall PLB hit rate of both HIVE and Flat ORAM. The large memory footprint of the HIVE's inverse position map structure results in overall more data being inserted into the PLB and hence translates into higher number of evictions from PLB. Consequently, the ORAM controller experiences higher number of PLB misses and issues relatively higher number of DRAM accesses, whereas the dense structure of OccMap offers a smaller memory footprint, thus causing less PLB evictions and exhibiting better locality (cf. Section 5.3), which directly translates into performance gain.

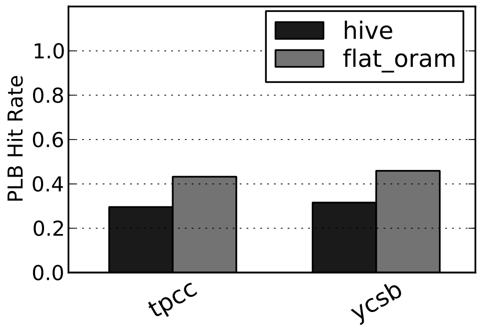
The normalized number of DRAM accesses is proportional to the energy consumption of the memory subsystem, i.e., a higher number of DRAM accesses would result in more energy consumption. On average, Flat ORAM saves up to 80% energy over HIVE for various workloads.



(**a**) Splash2



(**b**) SPEC06



(**c**) DBMS

Figure 3. Overall Position map Lookaside Buffer (PLB) hit rate (Position Map (PosMap) blocks and OccMap blocks).

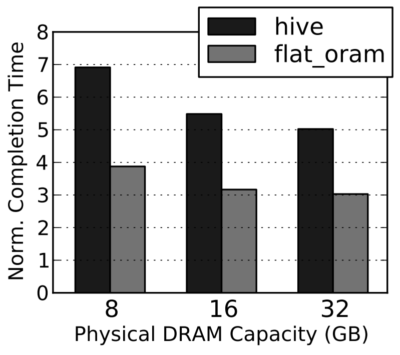
8.3. Sensitivity Study

In this section, we will study how different parameters in the system affect the performance of write-only ORAMs.

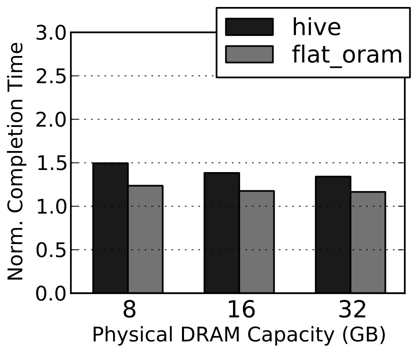
8.3.1. DRAM Utilization

When a block needs to be written to the DRAM, a random position is chosen and if that location is vacant, the block is written at that location (cf. Section 4.2). The probability that a randomly chosen location is 'vacant' is determined by the DRAM utilization, i.e., the ratio of occupied blocks to total blocks in DRAM. In order to study the effect of DRAM utilization, we show the results of various physical DRAM sizes (8, 16, 32 GB) for a constant working set of 4 GB in Figure 4. The resulting DRAM utilizations are 50%, 25%, and 12.5%, respectively.

Going from 50% to 25% utilization, memory intensive benchmarks (ycsb) gain performance, as the collisions during write operations are reduced by half. However, the jump from 25% to 12.5% utilization yields little gain because the collision probability of 25% at the 16 GB mark is already too low to be a major performance bottleneck. Notice that HIVE benefits more compared to Flat ORAM from the reduced collisions since it has a much higher collision-penalty. Since less memory intensive benchmarks (sjeng) are not constrained by write operations anyway, lower utilizations do not help much.



(**a**) ycsb



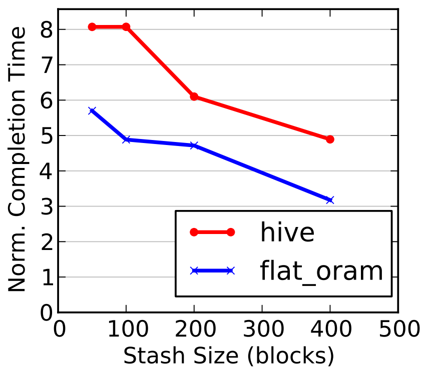
(**b**) sjeng

Figure 4. Sweep physical DRAM capacity.

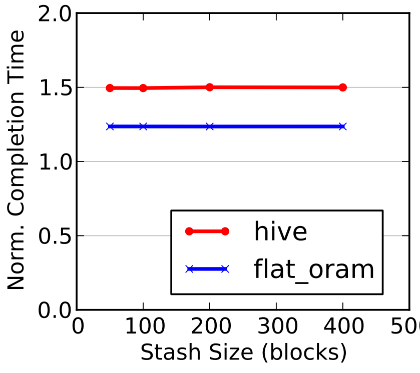
8.3.2. Stash Size

As discussed in Section 4.6, when the stash occupancy increases than a particular threshold, the ORAM starts performing 'background evictions'. Since background evictions cause the real requests to be suspended temporarily, frequent background evictions cause performance degradation. A larger stash is less likely to become full and thus reduces background eviction rate and improves performance.

In Figure 5, the stash size is swept for two different benchmarks, one is highly memory intensive (ocean_non_contiguous) and the other one is significantly less memory bound (sjeng). The memory intensive benchmark benefits from a large stash, as it experiences high background evictions rate at lower stash sizes. The less memory intensive benchmark does not benefit much from increased stash sizes, as it already has a low background evictions rate. In general, Flat ORAM shows significant performance gain over HIVE even at small stash sizes.



(**a**) ocean_non_contiguous

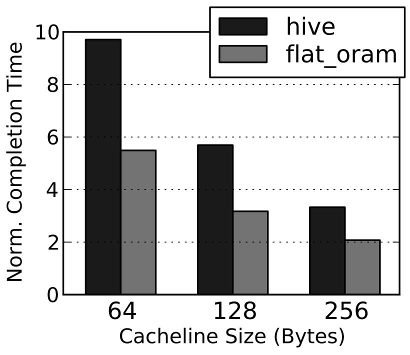


(**b**) sjeng

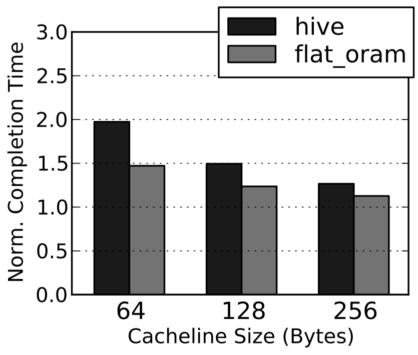
Figure 5. Sweep stash size.

8.3.3. Cacheline Size

We use the default cacheline size of 128 Bytes, in order to match the parameters in prior works [26,28,39]. Figure 6 shows the performance of HIVE and Flat ORAM for three different cacheline sizes (64, 128, and 256 Bytes). Having a larger cacheline size allows larger PosMap scale factors which results in overall less PosMap hierarchies (cf. Section 6.1). Additionally, memory intensive benchmarks benefit from large block sizes by exploiting data locality within blocks. This behavior is depicted in our results.



(**a**) ocean_contiguous



(**b**) sjeng

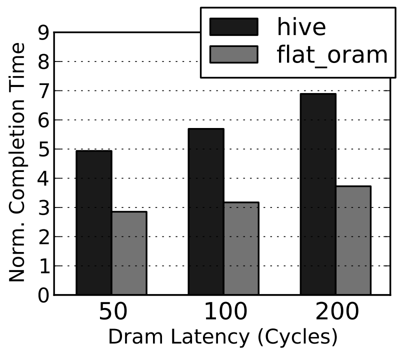
Figure 6. Sweep cacheline size.

8.3.4. DRAM Latency & Bandwidth

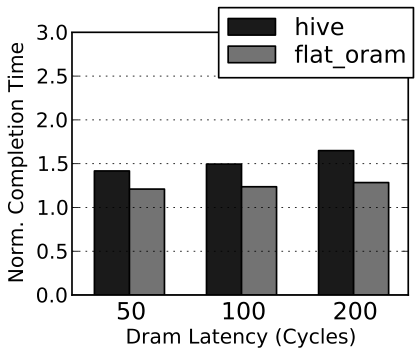
In Path ORAM, each ORAM access results in about 200 DRAM accesses on average under typical parameter settings. Most of these accesses can be issued in a burst without waiting for the first data block to arrive, since the addresses are known a priori, e.g., accessing a full path. Therefore, the DRAM bandwidth becomes the main bottleneck in Path ORAM, whereas the DRAM latency plays less significant role as it is incurred less often.

However, the write-only ORAMs under consideration typically only issue less than 10 DRAM accesses per ORAM access (cf. Section 8.2). Furthermore, there could be interdependencies within these 10 accesses, e.g., reading an OccMap block to find out if a position is vacant, and then issuing further writes in case a vacant position is found. In such cases, DRAM latency is incurred more often and hence plays more prominent role in the overall performance than the DRAM bandwidth.

This phenomenon is shown in DRAM latency and bandwidth sweep studies in Figures 7 and 8 respectively. Memory intensive benchmarks (ocean_contiguous) are more sensitive to DRAM latency and experience more performance degradation at higher latencies. On the other hand, compute bound benchmarks (sjeng) are less sensitive to the DRAM latency. Increasing the DRAM bandwidth seems to help only a little as expected and explained in the discussion above.

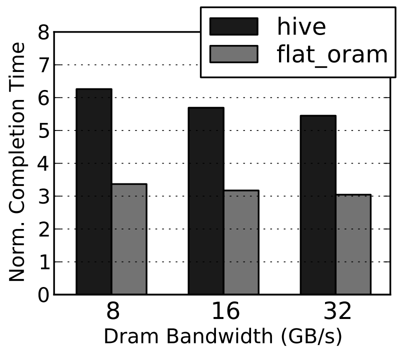


(**a**) ocean_contiguous

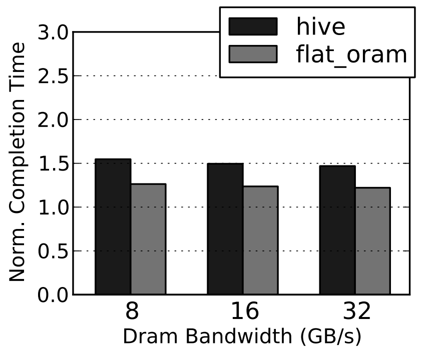


(**b**) sjeng

Figure 7. Sweep DRAM latency.



(**a**) ocean_contiguous

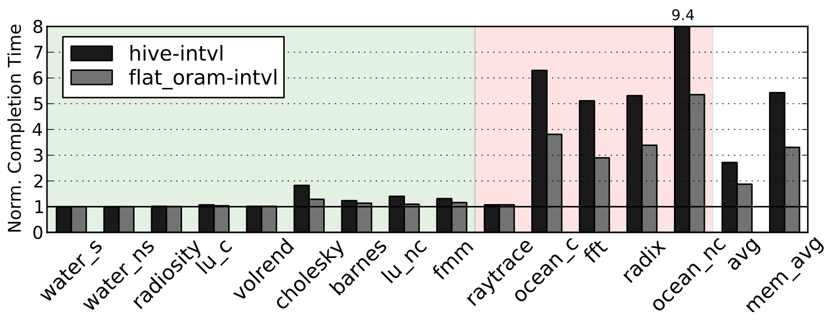


(**b**) sjeng

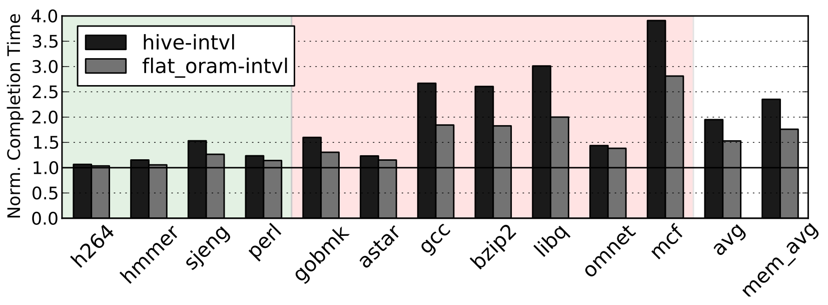
Figure 8. Sweep DRAM bandwidth.

8.3.5. Periodic ORAM

Figure 9 shows the experimental results of periodic write-only ORAM schemes normalized to insecure DRAM. The period in terms of number of cycles between two consecutive ORAM accesses is chosen to be 100 cycles. In general, adding periodicity to our ORAM scheme does not significantly hurt performance.



(**a**) Splash2



(**b**) SPEC06

Figure 9. Periodic Oblivious RAM (ORAM) accesses (100 cycles period). Normalized w.r.t. insecure DRAM.

9. Conclusions

We propose an efficient and practical write-only Oblivious RAM scheme called Flat ORAM for secure processor architectures. It is the first write-only ORAM with a concrete implementation in the secure processors domain. The implementation details are discussed and the design space is comprehensively explored. On memory intensive Splash-2 and SPEC06 benchmarks, Flat ORAM only incurs on average 3× and 1.6× slowdown, respectively. Compared to the closest related work in the literature, Flat ORAM offers up to 75% higher performance and 80% energy savings.

Author Contributions: Conceptualization, S.K.H., M.v.D.; Data curation, S.K.H.; Formal analysis, M.v.D.; Funding acquisition, M.v.D.; Investigation, S.K.H.; Methodology, S.K.H.; Project administration, M.v.D.; Resources, M.v.D.; Software, S.K.H.; Visualization, M.v.D.; Writing – original draft, S.K.H.; Writing – review & editing, S.K.H., M.v.D.

Funding: This work was partially supported by NSF grant CNS-1413996 for MACS: A Modular Approach to Cloud Security.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Arbaugh, W.; Farber, D.; Smith, J. A Secure and Reliable Bootstrap Architecture. In Proceedings of the 1997 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 4–7 May 1997; pp. 65–71.
2. Sarmenta, L.F.G.; van Dijk, M.; O'Donnell, C.W.; Rhodes, J.; Devadas, S. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In Proceedings of the 1st ACM Workshop on Scalable Trusted Computing (STC'06), Alexandria, VA, USA, 3 November 2006.
3. Trusted Computing Group. TCG Specification Architecture Overview Revision 1.2. 2004. Available online: https://trustedcomputinggroup.org/ (accessed on 24 March 2019).
4. Grawrock, D. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*; Intel Press: Hillsboro, OR, USA, 2006.
5. Lie, D.; Mitchell, J.; Thekkath, C.; Horwitz, M. Specifying and Verifying Hardware for Tamper-Resistant Software. In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 11–14 May 2003.
6. Lie, D.; Thekkath, C.; Horowitz, M. Implementing an Untrusted Operating System on Trusted Hardware. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, 19–22 October 2003; pp. 178–192.
7. Lie, D.; Thekkath, C.; Mitchell, M.; Lincoln, P.; Boneh, D.; Mitchell, J.; Horowitz, M. Architectural Support for Copy and Tamper Resistant Software. In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), Cambridge, MA, USA, 12–15 November 2000; pp. 168–177.
8. Suh, G.E.; Clarke, D.; Gassend, B.; van Dijk, M.; Devadas, S. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the 17th ICS (MIT-CSAIL-CSG-Memo-474 Is an Updated Version)*; ACM: New York, NY, USA, 2003.
9. Suh, G.E.; O'Donnell, C.W.; Sachdev, I.; Devadas, S. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05), Madison, MI, USA, 4–8 June 2005; ACM: New York, NY, USA, 2005.
10. Fletcher, C.; van Dijk, M.; Devadas, S. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing, Raleigh, NC, USA, 15 October 2012; pp. 3–8.
11. Maas, M.; Love, E.; Stefanov, E.; Tiwari, M.; Shi, E.; Asanovic, K.; Kubiatowicz, J.; Song, D. Phantom: Practical oblivious computation in a secure processor. In Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; ACM: New York, NY, USA, 2013; pp. 311–324.
12. McKeen, F.; Alexandrovich, I.; Berenzon, A.; Rozas, C.V.; Shafi, H.; Shanbhogue, V.; Savagaonkar, U.R. Innovative instructions and software model for isolated execution. In Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13), Tel-Aviv, Israel, 23–24 June 2013; p. 10.
13. Zhuang, X.; Zhang, T.; Pande, S. HIDE: An infrastructure for efficiently protecting information leakage on the address bus. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Boston, MA, USA, 7–13 October 2004; doi:10.1145/1024393.1024403.
14. Goldreich, O.; Ostrovsky, R. Software protection and simulation on oblivious RAMs. *J. ACM* **1996**, *43*, 431–437. [CrossRef]

15. Boneh, D.; Mazieres, D.; Popa, R.A. Remote Oblivious Storage: Making Oblivious RAM Practical. 2011. Available online: http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf (accessed on 24 March 2019).
16. Damgård, I.; Meldgaard, S.; Nielsen, J.B. Perfectly Secure Oblivious RAM without Random Oracles. In *Theory of Cryptography Conference*; Springer: Berlin/Heidelberg, Germany, 2011.
17. Goodrich, M.T.; Mitzenmacher, M.; Ohrimenko, O.; Tamassia, R. Oblivious RAM simulation with efficient worst-case access overhead. In Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW '11), Chicago, IL, USA, 21 October 2011; ACM: New York, NY, USA, 2011; pp. 95–100. [CrossRef]
18. Goodrich, M.T.; Mitzenmacher, M.; Ohrimenko, O.; Tamassia, R. Practical oblivious storage. In Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY '12), San Antonio, TX, USA, 7–9 February 2012; ACM: New York, NY, USA, 2012; pp. 13–24. [CrossRef]
19. Goodrich, M.T.; Mitzenmacher, M.; Ohrimenko, O.; Tamassia, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, Kyoto, Japan, 17–19 January 2012.
20. Ostrovsky, R. Efficient computation on oblivious RAMs. In Proceedings of the Nineteenth Annual ACM Symposium On Theory of Computing, New York, NY, 25–27 May 1987.
21. Ostrovsky, R.; Shoup, V. Private Information Storage (Extended Abstract). In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, El Paso, TX, USA, 4–6 May 1997; pp. 294–303.
22. Shi, E.; Chan, T.H.H.; Stefanov, E.; Li, M. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *Advances in Cryptology—ASIACRYPT*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 197–214.
23. Stefanov, E.; Shi, E.; Song, D. Towards practical oblivious RAM. In Proceedings of the NDSS Symposium 2012, San Diego, CA, USA, 5–8 February 2012.
24. Stefanov, E.; van Dijk, M.; Shi, E.; Fletcher, C.; Ren, L.; Yu, X.; Devadas, S. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In Proceedings of the ACM Computer and Communication Security Conference, Berlin, Germany, 4–8 November 2013.
25. Williams, P.; Sion, R. Single round access privacy on outsourced storage. In Proceedings of the 2012 ACM Conference on Computer And Communications Security (CCS '12), Raleigh, NC, USA, 16–18 October 2012; ACM: New York, NY, USA, 2012; pp. 293–304, doi:10.1145/2382196.2382229. [CrossRef]
26. Fletcher, C.; Ren, L.; Kwon, A.; van Dijk, M.; Devadas, S. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Istanbul, Turkey, 14–18 March 2015.
27. Zhang, X.; Sun, G.; Zhang, C.; Zhang, W.; Liang, Y.; Wang, T.; Chen, Y.; Di, J. Fork path: Improving efficiency of ORAM by removing redundant memory accesses. In Proceedings of the 48th International Symposium on Microarchitecture, Waikiki, HI, USA, 5–9 December 2015; ACM: New York, NY, USA, 2015; pp. 102–114.
28. Yu, X.; Haider, S.K.; Ren, L.; Fletcher, C.; Kwon, A.; van Dijk, M.; Devadas, S. Proram: Dynamic prefetcher for oblivious ram. In Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015; ACM: New York, NY, USA, 2015; pp. 616–628.
29. Merin John, T.; Kamran Haider, S.; Omar, H.; van Dijk, M. Connecting the Dots: Privacy Leakage via Write-Access Patterns to the Main Memory. *IEEE Trans. Dependable Secur. Comput.* **2017**. [CrossRef]
30. Joye, M.; Yen, S.M. The Montgomery powering ladder. In *International Workshop on Cryptographic Hardware and Embedded Systems*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 291–302.
31. Blass, E.O.; Robertson, W. TRESOR-HUNT: Attacking CPU-bound encryption. In Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, FL, USA, 3–7 December 2012; ACM: New York, NY, USA, 2012; pp. 71–78.
32. Böck, B.; Austria, S.B. *Firewire-Based Physical Security Attacks on Windows 7, EFS and BitLocker*; Secure Business Austria Research Lab: Vienna, Austria. 2009.
33. Maartmann-Moe, C. Inception. 2011. Available online: http://www.breaknenter.org/projects/inception/ (accessed on 25 February 2014).
34. Boileau, A. Hit by a bus: Physical access attacks with Firewire. *Present. Ruxcon* **2006**, *3*, 1–36.

35. Panholzer, P. Physical security attacks on windows vista. In *SEC Consult Vulnerability Lab Vienna Technology Report*; SEC Consult Vulnerability Lab: Vienna, Austria, 2008.
36. Stewin, P.; Bystrov, I. Understanding DMA malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 21–41.
37. Wojtczuk, R.; Rutkowska, J. Attacking intel trusted execution technology. *Black Hat DC* **2009**, *2009*, 1–6.
38. Wojtczuk, R.; Rutkowska, J.; Tereshkin, A. *Another Way to Circumvent Intel Trusted Execution Technology*; Invisible Things Lab: Warsaw, Poland, 2009; pp. 1–8.
39. Ren, L.; Yu, X.; Fletcher, C.; van Dijk, M.; Devadas, S. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In Proceedings of the 40th Annual International Symposium on Computer Architecture, Tel-Aviv, Israel, 23–27 June 2013.
40. Blass, E.O.; Mayberry, T.; Noubir, G.; Onarlioglu, K. Toward robust hidden volumes using write-only oblivious RAM. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, Arizona, USA, 3–7 November 2014; ACM: New York, NY, USA, 2014; pp. 203–214.
41. Haider, S.K.; van Dijk, M. Revisiting Definitional Foundations of Oblivious RAM for Secure Processor Implementations. *arXiv* **2017**, arXiv:1706.03852.
42. Fletcher, C.; Ren, L.; Yu, X.; Van Dijk, M.; Khan, O.; Devadas, S. Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-offs. In Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014.
43. Li, L.; Datta, A. Write-only oblivious RAM-based privacy-preserved access of outsourced data. *Int. J. Inf. Secur.* **2013**, *16*, 23–42. [CrossRef]
44. Roche, D.S.; Aviv, A.; Choi, S.G.; Mayberry, T. Deterministic, Stash-Free Write-Only ORAM. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), Dallas, TX, USA, 30 October–3 November 2017; ACM: New York, NY, USA, 2017; pp. 507–521. [CrossRef]
45. Miller, J.E.; Kasture, H.; Kurian, G.; Gruenwald, C.; Beckmann, N.; Celio, C.; Eastep, J.; Agarwal, A. Graphite: A Distributed Parallel Simulator for Multicores. In Proceedings of the HPCA—16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, Bangalore, India, 9–14 January 2010.
46. Woo, S.C.; Ohara, M.; Torrie, E.; Singh, J.P.; Gupta, A. The SPLASH-2 programs: Characterization and methodological considerations. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, S. Margherita Ligure, Italy, 22–24 June 1995; pp. 24–36.
47. Henning, J.L. SPEC CPU2006 benchmark descriptions. *ACM Sigarch Comput. Archit. News* **2006**, *34*, 1–17. [CrossRef]
48. Yu, X.; Bezerra, G.; Pavlo, A.; Devadas, S.; Stonebraker, M. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* **2014**, *8*, 209–220. [CrossRef]
49. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10), Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.
50. The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). 2007. Available online: http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp (accessed on 24 March 2019).