2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Shadow Block: Accelerating ORAM Accesses with Data Duplication

Xian Zhang*, Guangyu Sun*, Peichen Xie*, Chao Zhang*, Yannan Liu†, Lingxiao Wei†, Qiang Xu†, Chun Jason Xue‡
*Center for Energy-Efficient Computing and Applications, Peking University, Beijing, China
{zhang.xian, gsun, xpc, zhang.chao}@pku.edu.cn
†Department of Computer Science & Engineering, The Chinese University of Hong Kong, Hong Kong
{ynliu, lxwei, qxu}@cse.cuhk.edu.hk
‡Department of Computer Science, City University of Hong Kong, Hong Kong
jasonxue@cityu.edu.hk

***Abstract*—Oblivious RAM (ORAM) is a cryptographic primitive designed to hide memory access patterns. To achieve this objective, the intended data block is loaded and evicted back together with other data blocks and dummy blocks in each ORAM access. To further protect the timing pattern, extra dummy ORAM accesses are triggered periodically. Such designs lead to huge memory access overheads. Many techniques have been proposed to mitigate this problem by reducing the total number of ORAM accesses and the number of blocks per access. However, the impact of the access order of intended data block in an ORAM access is not addressed yet. In this work, we argue that higher performance can be achieved by advancing the access to the intended data block in ORAM accesses.**

However, changing the access order of blocks directly compromises the ORAM security. To solve this problem, we propose a duplication method to advance the access to the intended data blocks without compromising the ORAM security. The method leverages dummy blocks to store extra copies of data blocks, to facilitate early access of intended data blocks. These dummy blocks with valid data duplications are called Shadow blocks in this work. We further introduce two data duplication techniques, called RD-Dup and HD-Dup, to reorder the data block access for different purposes. In addition, we propose ORAM space partitioning to make RD-Dup and HD-Dup cooperate with each other efficiently. Compared with state-of-the-art ORAMs, our design can achieve a 32% reduction in system execution time on average, with negligible hardware overheads.

I. Introduction

To ensure a secure and private environment for program execution, memory encryption is widely proposed in various secure hardware, such as Trusted Computing Module (TPM) [1], eXecute Only Memory (XOM) [2], AEGIS [3] and etc [4]–[6]. However, it is insufficient to merely encrypt data because the data access pattern can also leak considerable sensitive information [7]–[9]. For example, the adversary can use the memory access addresses to construct the Control Data Flow Graph (CDFG) to infer program being executed. Even worse, the details of branches can be obtained with sophisticated skills to indicate secret keys [7].

To overcome this problem, Oblivious RAM (ORAM) is proposed as the general solution. ORAM is a cryptographic primitive that can completely hide the access patterns to memory [10]–[12]. A single data access is transformed into sequential accesses to memory blocks, which hold real program data, and **dummy blocks**, which hold useless data. Memory access maintains an encrypted and shuffled form for all data stored in the memory. For each memory access, data are re-encrypted and reshuffled. With the help of ORAM, any memory access pattern is computationally indistinguishable from others of the same length [13]–[15].

The main obstacle of using ORAM is its large memory access overhead. For every ORAM access, a number of dummy blocks and data blocks are loaded together with the intended data block. Before the access to the intended data block, a number of blocks are fetched to the processor, leading to significant increase in memory access latency and an average of $3\times \sim 4\times$ system-level performance degradation [11], [14], [15]. With timing protection, ORAM's overhead can be even higher because of additional ORAM requests [16], [17]. To tackle this problem, a number of techniques have been proposed in the literature [11], [14], [15], [17]–[19]. These approaches focus on reducing the total number of ORAM accesses and the number of blocks per access, leading to an order of reduction in access overhead.

However, prior researches do not consider the potential of reordering the block accesses within an ORAM request. In fact, we discover that the access order of the intended data block has a significant impact on ORAM performance. With earlier access to the intended data blocks, the interval between data requests or even the number of requests can be reduced (more details in Section III). Unfortunately, we cannot arbitrarily advance the data access. Accessing the intended data block first along the path disturbs the access pattern of the ORAM request, which compromises ORAM security.

To overcome this limitation, we propose that it is possible to leverage the dummy blocks, which normally occupy 50% or above of total storage space, to advance accesses to the intended blocks without security loss. To be specific, the access to an intended block can be advanced if we previously duplicate its data to a dummy block, which is read earlier in future ORAM accesses. We call such dummy blocks with data duplications as **Shadow Blocks**. Based on shadow blocks, we propose two duplication techniques named *Rear Data Duplication* (RD-Dup) and *Hot Data Duplication* (HD-Dup) to improve ORAM performance. Considering that RD-Dup

This work was supported by NSFC 61832020, NSFC 61572045, NSFC 61432017 and NSFC 61532017.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00082

and HD-Dup techniques may interfere with each other, we further propose ORAM partitioning techniques to make them cooperate with each other.

The main contributions are listed as follows,

- To the best of our knowledge, this is the first work addressing the impact of data access order on ORAM performance.
- We reveal that the access to an intended data block can be advanced using Shadow blocks without causing security loss.
- We propose RD-Dup and HD-Dup duplication techniques to leverage Shadow blocks so that ORAM performance is substantially improved.

The rest of this paper is organized as follows. We review the attack model and basic knowledge of ORAM in Section II. In addition, a state-of-the-art ORAM design called Tiny ORAM is described as the baseline of this work. Then, Section III illustrates motivation of this work. In Section IV, we introduce details of shadow blocks and propose RD-Dup and HD-Dup duplication techniques based on it. Moreover, static and dynamic partitioning schemes are also presented to make RD-Dup and HD-Dup cooperate with each other. We describe the detailed hardware design to enable the duplication techniques in Section V. A comprehensive evaluation of our design and comparison with the baseline are provided in Section VI. Section VII summarizes the related works, followed by a conclusion in the last section.

II. Preliminaries

In this section, the threat model is first presented. Then, ORAM basics are introduced to explain how the memory is protected from the attack. Last, Tiny ORAM is detailedly described as a state-of-the-art ORAM design.

A. Attack Models

Similar to the previous work [11], [14], [19], in our attack model, private programs are running on a secure processor. All data inside the processor are invisible to the outside. The processor is interacting with an untrusted external memory in an untrusted environment. All the information transported in the untrusted environment outside the processor will be exposed to the attacker. Not only the data content itself but also the access pattern of data should be protected. In addition, the timing of the access can be recorded by the attacker for further analysis [15], [16].

While data contents are protected by data encryption [2], [3], [6], [20], [21], the memory access patterns and timing channel can still leak considerable amount of secrets [7], [16]. This paper mainly focuses on the side channel leakage from the memory address access patterns and corresponding timing channel, which are widely discussed in ORAM designs [15]–[17], [19]. Other attacks are orthogonal to ORAM designs and out of scope for this paper, such as active attacks [20]–[22], convert channel attacks [23], [24], EM-attacks [25], [26], cache-timing attack [27], [28] and etc [29], [30].

B. ORAM Basics

ORAM is a cryptographic primitive designed to completely hide the memory access patterns. In ORAM, a single memory request sequence is transformed into a serial of memory accesses, which is called an ORAM request. As defined in previous work, an ORAM design is considered to be secure *if any two ORAM request sequences with the same length are computationally indistinguishable, no matter what their original memory request sequences are* [12], [13], [15].

Beside the core function to hide access patterns to external memory, **timing protection** is also widely proposed in previous ORAM designs [15], [17], [19], [31] and other related work [32], [33]. The timing channel of memory access can indicate information inside the secure processor. For example, the timing channel can imply when the cache miss occurs. Thus, program locality can be inferred by the attacker [16]. A data-independent and non-stop accesses to the external memory can eliminate this leakage [15], [16]. For the leakage from length of ORAM request sequence, since leakage bits grows logarithmically with the increment of the sequence length, this leakage can be omitted compared to the linear leakage from memory access pattern and timing channel leakage [14].

C. Tiny ORAM

In this section, we will introduce one of the state-of-art ORAMs called Tiny ORAM [18]. We use Tiny ORAM as an example because of its algorithm simplicity and implementation efficiency. Tiny ORAM is directly derived from the Path ORAM [13], which has been proposed and increasingly optimized [11], [14]–[16], [19]. Tiny ORAM is compatible to nearly all optimizations to Path ORAM, and can theoretically achieve a lower access overhead compared to Path ORAM [18]. In fact, optimizations in this paper can be applied to any other ORAMs that utilize dummy blocks, such as Ring ORAM [34], SSS ORAM [12] and etc [35]–[37].

Figure 1 (a) illustrates an overview of the Tiny ORAM architecture, which includes two components: (1) an untrusted external memory (above CPU-memory boundary) and (2) a trusted on-chip ORAM controller (below CPU-memory boundary), which are described as follows:

The external memory is logically structured as a binary tree called **ORAM tree** [14], [15]. As shown in Figure 1 (a), $L+1$ levels are contained in the ORAM tree, which are denoted as level 0 (root), level 1, ..., level L (leaf). The node of the tree is called a *bucket*, which holds a fixed number (denoted as Z) of slots to store memory blocks. One bucket can hold $0 \sim Z$ **data blocks**, which hold the valid data of the program. If the bucket is not full, the rest of a bucket is filled with **dummy blocks**, which hold meaningless data for confusion. Both data blocks and dummy blocks are probabilistically encrypted with One Time Pad [4], [5], [11]. Thus, any two cipher blocks are indistinguishable, even when their plaintext are the same and no matter they are dummy or data blocks. Every leaf node in the ORAM tree is assigned a label ranging from 0 to $2^L - 1$. And path-l is defined as the path from the leaf with label l to the root. For instance, as shown in Figure 1 (a) and Figure 1 (c), path-2 is highlighted.

The ORAM controller includes two main components: a *stash* and a *position map* (denoted as "PosMap" in Figure 1). The stash is an on-chip memory component to temporarily hold a small number (e.g. 200 data blocks [11], [14], denoted as M) of data blocks. The position map is a lookup table recording the leaf labels for every program address. At runtime, each data block is assigned with a random leaf label. Tiny ORAM design follows an **invariant** [11], [13]: *a data block with a leaf label of l must be either in the stash or path l.* Data blocks are always stored together with their program

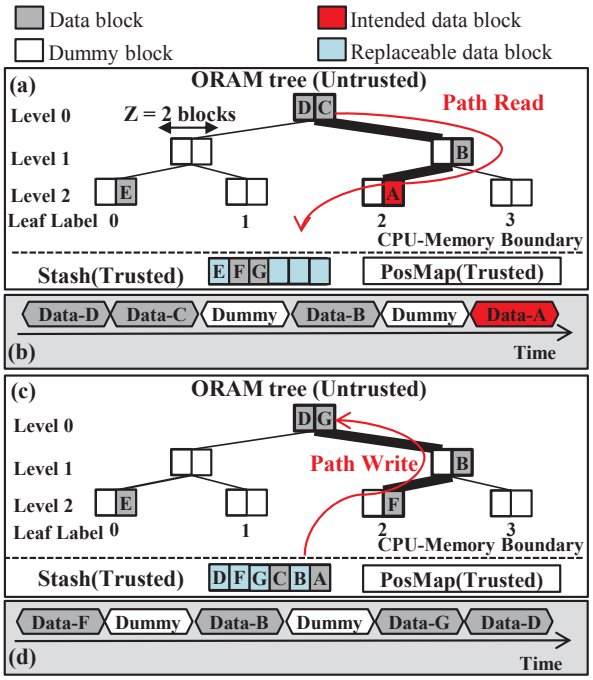

Fig. 1. Illustration of path read and path write with $L = 2$, $Z = 2$ and $M = 6$ (a) Path read (b) Timing diagram of path read (c) Path Write (d) Timing diagram of path write

addresses (namely the query addresses from CPU) and leaf labels, regardless of whether they are in the stash or in the external memory.

For a memory request denoted as $(addr, op, data)$, where $addr$ represents the program address and op denotes the operation type (read/write), it is transformed into an ORAM request in following steps [15], [18]:

- **Step-1:** Stash is queried for a data block with $addr$. If the data block is present, it is forwarded to CPU Last Level Cache (LLC).
- **Step-2:** If a stash miss occurs, ORAM controller refers to the position map for the leaf label (l) of $addr$.
- **Step-3 (Path Read):** Blocks along path-l are loaded from ORAM tree and stored into stash after decryption. During the path read, the intended data block required by the processor is forwarded to LLC for further read or write, while dummy blocks are all discarded. Then, the leaf label for $addr$ is remapped to l' and the position map is updated. During path read, data blocks along the path-l are invalidated.
- **Step-4:** Path read operations are launched a fixed (A-1) times. Number A is normally called eviction rate.
- **Step-5:** Path-k is chosen following a reverse lexicographical order [18], [34]. Then, path-k is loaded through a path read.
- **Step-6 (Path Write):** Path k is re-filled with data blocks in the stash. The principle is to re-fill the path with data blocks in stash "as many as possible" [11], [13] following Path ORAM invariant. Data blocks written back to memory from the stash are marked as replaceable blocks, which means their corresponding positions in the stash become free slots to hold new data. Dummy blocks are inserted if there are free slots in the path.

In previous literature [18], Step-3 are called Read-Only phase, which actually loads the memory data to serve LLC read/write. Step-5 and Step-6 are called Read-Write phase, which aims at evicting blocks in the stash. Path read (Step-3, Step-5) and path write (Step-6) are the core steps in Tiny ORAM. An example is shown in Figure 1 to further illustrate them.

In Figure 1 (a), path-2 is loaded to access Data-A, which is intended. As shown in Figure 1 (b), before the access to the Data-A, three data blocks and two dummy blocks have to be accessed. Before the path read, block-E, F, G are in the stash. In addition, Data-E are replaceable in the stash since it has been evicted before. During the path read, Data-E and other replaceable blocks are replaced by incoming data blocks, while dummy blocks are discarded and not inserted into the stash. After the path read, the stash consists of Data-D, F, G, C, B, A. As shown in Figure 1 (c), during path write, block-D, F, G, B are evicted back and marked as replaceable. If there are no proper blocks in the stash to fill free slots in the path, dummy blocks are written, which is illustrated in Figure 1 (d).

Tiny ORAM can be further improved with techniques to address other design issues, such as proper configurations to ensure negligible stash overflow possibility [11], unified program address space to address external position map issue [14], and constant-rate requests to protect timing channel [16]. In the rest of this paper, a Tiny ORAM combined with these techniques is used as our baseline and denoted as Tiny ORAM for simplicity.

III. Motivation of Advancing Access

In this section, we discuss the potential benefits of advancing the intended data block access, which motivates this work.

In an ORAM request, before the intended data block is really accessed, a number of blocks including unneeded data blocks and dummy blocks have to be fetched by ORAM controller. This process can severely stall the program execution in CPU. Thus, if the access to intended data block is advanced, it is straightforward that the stalled CPU can be restored earlier. Then, the following ORAM requests can be issued earlier so that the overall memory performance is improved.

We use the code in Figure 2 (a) as an example. In this example, we assume that both Data-1 and Data-2 are not cached in CPU and need to be accessed in the ORAM tree. From the code, we can find that the CPU request for Data-2 depends on the result of Function-1(Data-1). Thus, if we can advance the access to Data-1 from the tail to the head of the ORAM request, ORAM request for Data-2 can be issued earlier, respectively. The effect is illustrated in Figure 2 (b) and (c). We can find that the data request interval (DRI) is reduced.

The benefits are more significant when there exists timing protection. As shown in Figure 2 (d), an extra dummy ORAM request is induced due to the long DRI between Data-1 and Data-2. As shown in Figure 2 (e), if we can advance the access time to Data-1, the dummy request can be even saved. Thus, the data request interval is further reduced more significantly.

From these two examples, we observe that *advancing the access to intended data block in an ORAM request can reduce the DRI between this ORAM request and the following one*. In fact, the DRI can occupy up to 50% of total execution time [16], especially when timing protection is equipped. Thus, the ORAM performance can be significantly improved if we

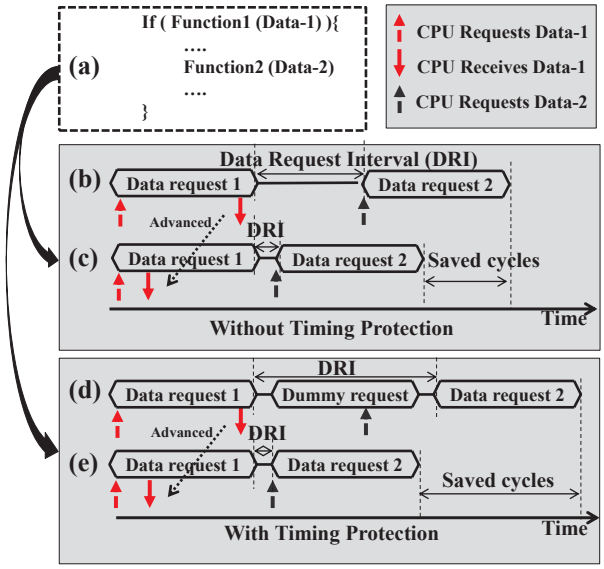

Fig. 2. Motivation illustration:(a) example code (b) original timing diagram (w/o timing protection) (c) timing diagram after advancing access to Data-1 (w/o timing protection) (d) original timing diagram (with timing protection) (e) timing diagram after advancing access to Data-1 (with timing protection)

can advance the access to intended data block in each ORAM request properly.

Unfortunately, the access to intended data cannot be advanced arbitrarily because of two obstacles. First, the exact position of the target block is unknown before the block is loaded into the stash along its loading path. Second, even the position is known before the access, we cannot directly change the access sequence because it will compromise the security of ORAM. We illustrate this as follows:

In an extreme case, we suppose that the intended block is always accessed first along the path. Thus, the attacker knows where the intended block locates in every path access. And there are two sequences of program addresses:

- Sequence$_1$ (Scan accesses): $\{a_1, a_2, ..., a_N\}$
- Sequence$_2$ (Cyclic accesses):$\{a_1, a_2, ..., a_k, a_1, ..., a_k, a_1, ...\}$

Here N represents the number of blocks in a memory (e.g. $N = 2^{26}$ for Table I) and $k \ll N$. If the intended block is observed to appear at a path that is written within last k path writes, we call this Read-Recent-Written-Path (RRWP-k). It is obvious that for Sequence$_2$, RRWP-k occurs more frequently than that of Sequence$_1$. Thus, Sequence$_1$ and Sequence$_2$ can be distinguished, which proves that changing the access sequence compromises the security of ORAM.

To overcome these obstacles, we propose an efficient and secure data duplication technique. This is inspired by the fact that there exist abundant dummy blocks, which normally occupy about 50% of memory space [11], [14], [18]. The basic idea is to store copies of data blocks in these dummy blocks without being aware by attackers. Thus, the access to intended data blocks can be advanced without any security loss. The detailed method is introduced in the next section.

IV. Data Duplication in ORAM

In this section, we first demonstrate how to advance data access through data duplication without compromising ORAM security. Then, Rear Data Duplication (RD-Dup) and Hot Data Duplication (HD-Dup) are proposed for two types of data blocks, respectively. Last, we introduce how to make RD-Dup and HD-Dup cooperate with each other under memory partitioning schemes.

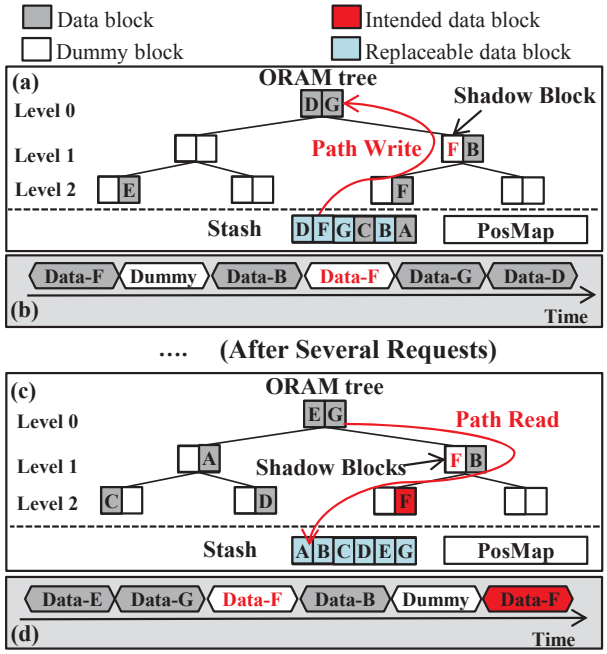

Fig. 3. An illustration of duplication: (a) Modified path write to duplicate data-F (b) Timing diagram of modified path write (c) Modified path read to read duplicated data (d) Timing diagram of modified path read

A. Shadow Blocks for Duplication

As discussed in last section, the attackers should be unaware of the duplication process to prevent security loss. Thus, the duplication cannot be performed by explicitly moving data in main memory. Instead, the duplication process is seamlessly integrated with the path write operation and completed inside the ORAM controller.

An example of duplication is shown in Figure 3 (a) for the Tiny ORAM. Compared to the original path write in Figure 1 (c), the only difference is that dummy blocks are filled with Data-F rather than some useless data. This dummy block is called a **Shadow block** of Data-F, which is highlighted in the figure. Since all blocks are re-encrypted in each path write, the shadow block is indistinguishable from the original dummy block.

A shadow block can be generated as long as the following three rules are satisfied:

- **Rule-1:** shadow block also follows the invariant of Tiny ORAM block described in Section II-C.
- **Rule-2:** shadow block always appears at lower levels of ORAM tree than the data block being duplicated.
- **Rule-3:** shadow block are replaceable in the stash and can be replaced by any incoming data blocks.

Rule-1 and Rule-2 guarantee that there is only one version of data for different copies in the stash and ORAM tree. Whenever the original data block is loaded to stash, its shadow block(s) (if existed) are always loaded together. Then, no matter the data block are updated or re-mapped to another

path, the shadow block(s) are always consistent with the data block. The stale shadow blocks are invalidated in the path read (Step-3 in Section II-C). Rule-3 guarantees that the stash overflow possibility is not affected, which is proved in the next subsection.

In a later path read operation, the Data-F is the intended data block. As shown in Figure 3 (c), the shadow block of Data-F is identified during before the real Data-F is loaded. Thus, the CPU can obtain requested data in shadow blocks and be restored earlier from stall. The timing diagram is shown in Figure 3 (d). Apparently, the access to Data-F is being advanced with the help of its shadow block. In order to identify the shadow blocks, the data structure of blocks is changed slightly. The only difference is that an additional bit is added to every block to indicate whether it is shadow block or not (referred as "shadow bit"). Namely every block in the ORAM tree or stash is in the form: $(shadowbit, data, label, addr)$ as shown in Figure 7(a).

A merge operation is introduced to handle multiple copies of data in stash. There are two cases of merge operations. First, if the original data block is loaded together with its shadow block(s) into stash, all shadow block(s) are discarded after the original data block is loaded. Second, if only multiple shadow blocks are loaded into the stash, they are merged as a single shadow block.

B. Security Proof of Data Duplication

In this section, we first prove that our scheme is as secure as Tiny ORAM in the aspect of memory access patterns. Then, we prove that the stash overflow possibility of our design and Tiny ORAM are also the same.

1) Access Pattern Security: For clarity, we list the pseudocodes of the path write and the path read in Algorithm 1 and Algorithm 2, respectively. $Path[l][i]$ represents the i_{th} block along the path with leaf label l. Compared to the path read and the path write of Tiny ORAM, modified operations in our design are highlighted with red color. We also highlight the operations that the CPU interacts with the external memory in blue color (Line 6 in Algorithm 1 and Line 2 in Algorithm 2). In other words, except operations in blue color, the rest operations all occur within the CPU and are **invisible** to attackers.

Algorithm 1: Path Write with Duplication

```
   Input: Leaf label l
1  for ( i =Z*L-1 ; i ≥ 0; i−−) do
2      blk ← stash_blk_select( );    /* select data from stash to be
       evicted to Path[l][i] */
3      if blk.type is dummy then
4          blk ← dup_blk_select();          /* select data to be
           duplicated to Path[l][i] */
5      end
6      Path[l][i]← Enc(blk);    /* Blk is encrypted and written to
       Path[l][i] in the memory */
7  end
```

Algorithm 2: Path Read with Duplication

```
   Input: Leaf label l
1  for ( i =0 ; i ≤ Z*L-1; i++) do
2      blk ← Dec(Path[l][i]);       /* Path[l][i] is read from the
       memory and decrypted */
3      if blk.type is real or blk.type is shadow then
4          stash_insert(blk);    /* blk is inserted into the stash
           */
5          if blk.addr is LLC_request_addr then
6              LLC← blk.data;    /* data of blk are forwarded to
               LLC */
7          end
8      end
9  end
```

Because the lines in blue color are common for both our design and Tiny ORAM, the interactions between the CPU and the memory of our scheme and Tiny ORAM are the same. The only change is that we change the content of blocks in path write (Line 4 in Algorithm 1) and we do not discard data of shadow blocks in path read (Line 3 and Line 4 in Algorithm 2). However, these internal operations cannot be observed by the attacker and leak no information. For the external activities that CPU interacts with the memory, the attacker cannot distinguish our scheme with Tiny ORAM because the probabilistically encryption is adopted. According to the definition of ORAM, our scheme is as secure as Tiny ORAM.

2) Stash Overflow Possibility: Similar to Path ORAM, one critical problem of Tiny ORAM is the stash overflow possibility [11], [14], [15]. In our design, real blocks can always replace the shadow blocks in the stash (Rule-3 in Section IV-A). Thus, the security strength of utilizing shadow blocks is the same to Tiny ORAM on the aspect of stash overflow possibility. Note that, if a stash size is set to ensure enough security parameter (i.e. negligible possibility of stash overflow), the stash can always hold all the data from a path, including the data blocks and shadow blocks [15].

Data duplication provides the potentials to utilize dummy blocks in the ORAM tree without compromising security. In the following sections, we illustrate two techniques based on data duplication: Rear Data Duplication (RD-Dup) and Hot Data Duplication (HD-Dup).

C. Two Duplication Schemes

As discussed in subsection IV-A, a shadow block can be generated as long as those three rules are satisfied. Thus, an critical problem is how to select a proper candidate data block to be duplicated for specific purpose. In this subsection, we propose two duplication schemes with different selection methods.

1) Rear Data Duplication: Since an intended data block in higher levels of ORAM tree can result in longer DRI, an intuitive method is to select the candidate with highest level. To simplify the discussion, we call such a candidate with highest level **Rear Data** in this work. And such a duplication method is called Rear Data Duplication (RD-Dup).

An example of write path is shown in Figure 4 to illustrate the RD-Dup. The conventional Tiny ORAM without duplication is shown in Figure 4 (a) for comparison. We assume that there are only four data in the stash, which are Data-A, B, C, D. Their positions in the Tiny ORAM tree are also shown in the figure after the path write.

The state of Tiny ORAM tree after using RD-Dup is shown in Figure 4 (b). When we write to the dummy block in Level-1, both Data-A and Data-B satisfy the rules of becoming shadow blocks. We assume that Data-A is loaded after the Data-B, though they are in the same bucket. Then, Data-A is selected for duplication and stored in the dummy block[1]. Note that the level of Data-A has **changed** to level-1 after duplication.

[1]If all blocks in a bucket are loaded concurrently, a block can be selected randomly [11].

Later, when we write to the dummy block in Level-0, Data-B is duplicated because it has highest priority.

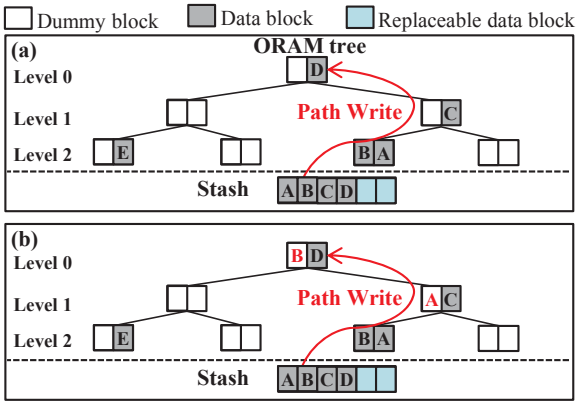

Fig. 4. (a) Path write without duplication (b) Path write with RD-Dup

2) Hot Data Duplication: The shadow blocks can also be leveraged to duplicate hot data that are frequently accessed. Since blocks located at lower levels have a higher probability to be loaded into the stash in a path read, the shadow blocks in low levels can help to cache hot data into stash. If those blocks contain the data to be used in the future, extra data requests can be avoided. Such a duplication method is called Hot Data Duplication (HD-Dup) in this work.

The basic work flow of Hot Data Duplication (HD-Dup) is the same as RD-Dup. The differences are listed as follows. The priority of a candidate block is defined as the access count of the block instead of its ORAM tree level. The priority of the blocks are stored in a Hot Address Cache (see SectionV for details), instead of the stash. When a shadow block is generated in a path write, HD-Dup searches the Hot Address Cache to identify the candidate with highest priority. Note that if a candidate is not in the access counter cache, priority of this block is set to zero.

D. Cooperating RD-Dup & HD-Dup

Since RD-Dup and HD-Dup have different duplication methods, they will interfere with each other. To mitigate this problem, we propose to partition the ORAM memory tree into two regions, which use RD-Dup and HD-Dup separately.

The rationale behind partitioning is explained as follows. Using blocks at low levels for RD-Dup is less efficient if the promoting time exceed DRI too much since next data request has to wait for the end of current request (see Figure 2 for intuitions). In addition, using HD-Dup is more efficient to duplicate hot data to blocks at the lower levels of ORAM tree because of frequent accesses. Therefore, allocating blocks at higher levels for HD-Dup is less efficient compared to allocating blocks to RD-Dup. Thus, there should be an optimal boundary to partition the ORAM tree into two parts: higher part and lower part. The level separates two parts is defined as the **Partitioning Level**. For dummy blocks above and in the partitioning level, shadow blocks are generated with RD-Dup, while the HD-Dup is employed in the lower part. A path write with partitioning is illustrated in Figure 5.

1) Static Partitioning: It is straightforward to apply partitioning with a fixed partitioning level for all programs. However, the optimal selection differs with programs and periods (see Section VI), which motivates the dynamic partitioning.

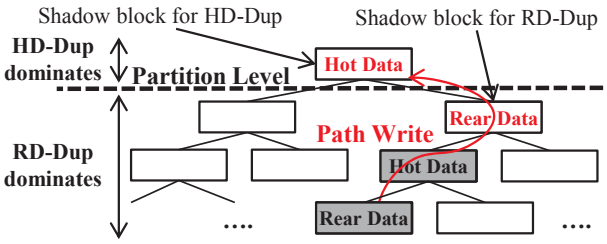

Fig. 5. Partitioning schemes to combine RD-Dup and HD-Dup

2) Dynamic Partitioning: While a fixed partitioning level simplifies the system design, it is not optimal for programs having varying LLC miss intervals. HD-Dup is more useful when programs have higher locality, where it needs higher partition level. RD-Dup becomes more useful if programs have longer LLC miss intervals. Unfortunately, locality and LLC miss intervals of programs are changing along the time.

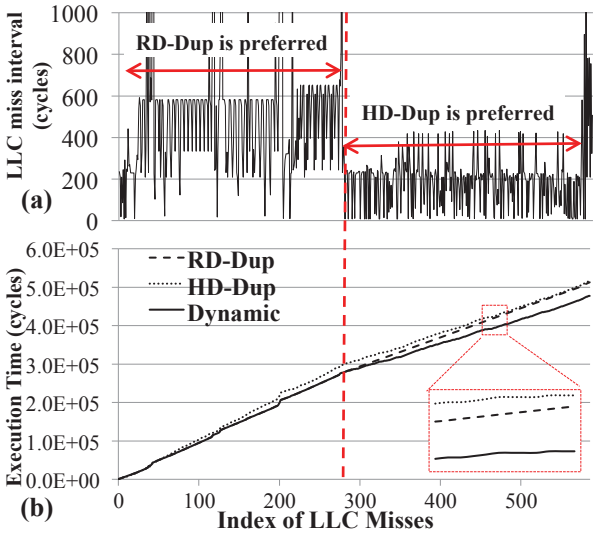

Fig. 6. (a) Sampled miss intervals for *hmmer* and (b) Corresponding execution time using different duplication schemes

As an example, a snippet of *hmmer* is illustrated in Figure 6 (a) to show the period-to-period variation of LLC miss intervals. The LLC misses are indexed with their order of appearance starting from 0. Intuitively, when most DRIs are small, HD-Dup is more efficient so that the partitioning level should increase. When most DRIs are large, the partitioning level should decrease to enable more RD-Dup duplication. We further illustrate the system execution time with HD-Dup and RD-Dup in Figure 6 (b), in which the curve gradient can represent the runtime latency to serve a single LLC miss. For the first half LLC misses, RD-Dup outperforms the HD-Dup whereas for the second half, HD-Dup is more efficient than RD-Dup with a lower curve gradient. This coincides with previous intuitions. Therefore, to benefit from both schemes, dynamically adjusting partitioning level is preferred.

Fortunately, the variation of miss intervals occurs periodically, which has been also shown in Figure 6 (a). Thus, we use a method similar to saturating counter [38] to reflect and predict the intensity of long LLC miss intervals. This

counter is called Data Request Interval Counter (DRI Counter). The counter updates after each ORAM request. If the current request is a dummy request and its previous request is a real request, the counter is increased by one. If the current request is a real request and its previous one is also a real request, the counter is reduced by one. Otherwise, the counter keeps unchanged. The strategy is based on the observation that a real request followed by a dummy request means the DRI is too long and RD-Dup is preferred. A real request followed by another real request means short DRIs and HD-Dup is preferred. The counter becomes saturated after it reaches zero or the maximum value ($2^{counter\_length} - 1$). If DRI Counter is smaller than the half of the maximum value, the partitioning level is increased by one level, and vise versa. Figure 6 (b) shows the execution time of a dynamic partitioning scheme with a 3-bit DRI counter. We can find a reduction in execution time w.r.t HD-Dup or RD-Dup.

It is noticeable that dynamic partitioning does not degrade the security of ORAM. For ORAM schemes without timing protection, the timing channel leakage of dynamic partitioning scheme is the same with that of a naive scheme. Both of them leaks information about the locality and LLC miss intervals of the protected programs. For ORAM schemes with timing protection, the dynamic scheme is as secure as other work [17], [19] and only leaks bits which grow logarithmically with the increment of overall program execution time [16].

E. Compatibility with ORAM Optimizations

In this section, we discuss the compatibility of shadow block to state-of-the-art ORAM optimizations and countermeasures to mitigate other attacks towards ORAM. Since shadow block only changes the content of dummy blocks, shadow block can be combined with almost any other ORAM optimizations, such as super block prefetching [11], [17], PosMap Lookup Table [14], treetop caching [15], fork path [19] and etc. In addition, shadow block can be combined with other countermeasures such as timing side channel protection [39], [40]. The only exception is XOR compression [12], [31], [34].

XOR compression is a state-of-the-art ORAM optimization, in which all blocks along a path are first XORed, and then only one block of the XOR result is sent to the processor. Thus, the bandwidth of CPU-memory bus is saved. However, there are two limitations of XOR compression. First, XOR compression requires that the memory has the capability of computing, which is not supported in most of today's commercial DRAM. Second, XOR compression has limited effect in reducing ORAM access latency. Actually, the bottleneck of ORAM is not at the CPU-memory bus, but at the internal bandwidth of DRAMs. Even if the processor-memory bandwidth is saved, the internal bandwidth still limits the access speed. We will compare XOR compression technique and shadow block technique in Section VI-C.

V. Hardware Design

In this section, we introduce the hardware modification to support Shadow blocks, RD-Dup, HD-Dup, and partitioning.

A. Stash Modification

As designed in previous work [15], the stash is implemented using a content addressable memory (CAM). Data with certain program address can be referred immediately. An evicted bit is used to denote a block is replaceable (a.k.a. replaceable block). The modification is as follows. First, a shadow block is also label as replaceable block after it is loaded into stash. Second, as mentioned before, merge operations can occur if two blocks in the stash have the same program address.

B. New Components in ORAM Controller

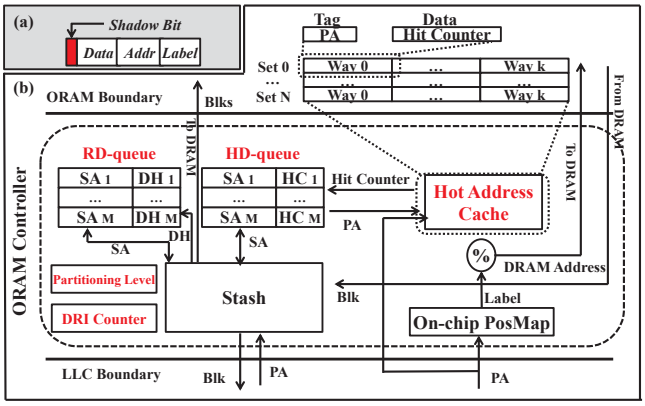

Fig. 7. Architectural support for Shadow block:(a) block structure (b) the structure of ORAM controller

To support HD-Dup and RD-Dup, three several components are added, which include Hot Address Cache, RD-queue, HD-queue, partitioning level register, and DRI counter register.

1) Hot Address Cache: Hot Address Cache stores the access counter for hot data. We implement it with a set-associative cache. The tag represents the program address and a Least Frequently Used (LFU) policy [41] is adopted. Note that this cache only store the program addresses from the LLC misses (read or write). Ever since a cache-line is hit in the cache, the corresponding counter will be increased by one.

2) RD-queue and HD-queue: Two queues called Rear Data queue (RD-queue) and Hot Data queue (HD-queue) are added to store all candidate data blocks that can be duplicated in RD-Dup and HD-Dup, respectively. RD-queue are used to sort the data levels of blocks in the queue. HD-queue is sorted according to the access counters of the candidate blocks.

Whenever a data block is evicted and written back to the ORAM tree, the stash address of that block will be inserted into this queue. Note that shadow blocks in the stash, which can be evicted, are also inserted into the queues. For HD-queue, if the inserted data is not found in the Hot Address Cache, the access counter of this data will be initialized as zero. When a dummy block is encountered during the path write, blocks at the head of these queues will be used to generate to the Shadow block. Both queues are cleared after the path write is completed.

3) Extra Registers: We add a partitioning level register to store the partitioning level. A DRI counter register is also needed to store the current DRI counter and update the partitioning level. For each dummy block to be filled, its level is compared with the partitioning level to select which duplication method (e.g. queue) to use.

C. Design Overhead

We evaluate the design overhead of Shadow block technique. For storage overhead, the main overhead comes from

the shadow bit (1-bit) in every block, which is approximately $4MB$ at DRAM. The Hot Address Cache is set to be $1KB$ in this work and thus introduces negligible overhead. For the logic overhead from structures described previously, circuit level synthesis results show that the main cost comes from the HD-queue and RD-queue. They require about $13,000$ gates, which is negligible compared with the area of ORAM logic.

VI. Evaluation

In this section, we first present the experimental setup. Then, we evaluate our design using a representative system configuration, in both scenarios with and without timing protection, respectively. In addition, we will show the evaluation results compared to previous work. Finally, sensitivity analysis of different configurations is provided.

A. Experimental Setup

To evaluate the performance of our designs, a full system simulator gem5 [42] integrated with DRAMSim2 [43] is adopted in this work. Table I lists the detailed configuration of processor, ORAM controller, and main memory. Latencies of ORAM control logic and cache are generated from Synopsys [44] and CACTI [45], respectively. DRAMSim2 [43] is used to model the latency of a ORAM path access. Default parameters of DDR3 latency from DRAMSim2 are adopted in the evaluation. Power consumption are evaluated with energy parameters from [16]. We also use a state-of-the-art configuration of O3 CPU [19] for the evaluation in Section VI-E. The O3 CPU and the in-order CPU only differs in core type/number and L2 cache, which are listed in Table I.

Two memory channels are employed as same as the typical configuration [11], [14]. To ensure a low probability of stash overflow, a 50% memory utilization is assumed in this work [11]. It means that to store a 4GB data, a 8GB DRAM is required. The block slots per bucket (Z) and eviction rate (A) are also set as $(5, 5)$, which is suggested in [18]. To fully tap the potential of DRAM bandwidth, a sub-tree layout is derived [11]. In order to make a comparison with previous work [18], ten workloads are selected from SPEC 2006 [46] benchmark suites for a comprehensive evaluation.

B. Evaluation Results without Timing Protection

In this section, we first evaluate the efficiency of RD-Dup and HD-Dup, relatively. Then, the static partitioning and dynamic partitioning schemes will be evaluated. Last, evaluation of performance and energy is present. To better illustrate the effect of HD-Dup and RD-Dup, we list the

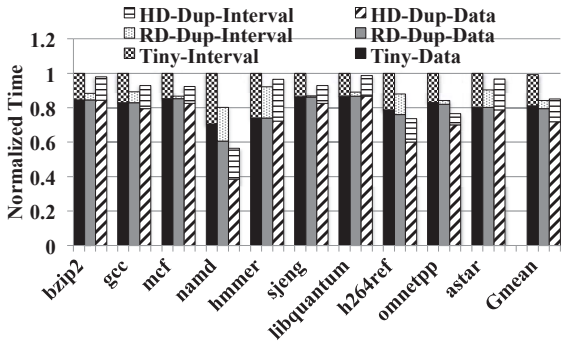

Fig. 8. Normalized access time using RD-Dup and HD-Dup, respectively

TABLE I
Processor and Memory Configuration.

Core, on-chip cache (in-order CPU)	
Core type	in-order single-core Alpha
Core frequency	2GHz
L1 I/D cache	32KB/32KB, 2-way, LRU
L1 read/write	1/1-cycle
L2 cache	1MB, 8-way, LRU
L2 read/write	10/10-cycle
Core, on-chip cache (O3 CPU [19])	
Core type	out-of-order Alpha
Core number	4, 8-way issue
L2 cache	1MB shared, 8-way, LRU
ORAM controller	
Controller clock frequency	2.0GHz
Data block size	64B
Data ORAM capacity	4GB (L = 24)
DRAM utilization	50%
PLB	64KB [14]
Block slots per bucket (Z)	5 [18]
Eviction rate (A)	5 [18]
AES-128 latency	32 cycles [14]
Memory controller and DRAM	
Memory type	DDR3-1333
Memory channels	2
Peak bandwidth	21.3GB/s

data access time and data request interval (DRI), respectively. Data access time means the time consumed by data ORAM requests. For the total execution time, we have:

$$Total\_execution\_time = Data\_access\_time + DRI \quad (1)$$

Figure 8 illustrates the data access time and DRI normalized to the total execution time of Tiny ORAM, respectively. The bar of Tiny-Data denotes the normalized data access time while Tiny-Interval denotes the normalized DRI. The bars of Tiny-Data and Tiny-Interval are stacked to represent the total execution time. The bar of (HD)RD-Dup-Data/-Interval represents the normalized data access time/DRI when (HD) RD-Dup is applied. We can see that RD-Dup mainly reduces the DRI while HD-Dup mainly reduces data access time. This agrees with the purposes of RD-Dup and HD-Dup. On average, RD-Dup reduces 74% DRI, 2% data access time and 16% total execution time compared with Tiny ORAM, respectively. By contrast, HD-Dup reduces 27% DRI, 12% data access time and 15% total execution time compared with Tiny ORAM, respectively.

Figure 9 illustrates the execution time of three representative benchmarks and ten workloads's geometric mean, after the static partitioning is applied. We investigate effects when the partitioning level increases from zero to 25. The Interval/Data curve represents the DRI/data access time normalized to the total execution time of Tiny ORAM. And the normalized total execution time is also illustrated. For most workloads, with the increase of partitioning level, the data access time decreases while the DRI increases. This is natural since more/less dummy blocks are assigned to HD-Dup/RD-Dup if the partitioning level is higher. One exception is $namd$, since the number of data requests is largely reduced, the DRI also reduces. For $sjeng$, the reduction of data access time is less than the increase of DRI, which leads to a increase in total execution time. And vice versa for $h264ref$. From the figure of geometric mean, we find that the total execution time

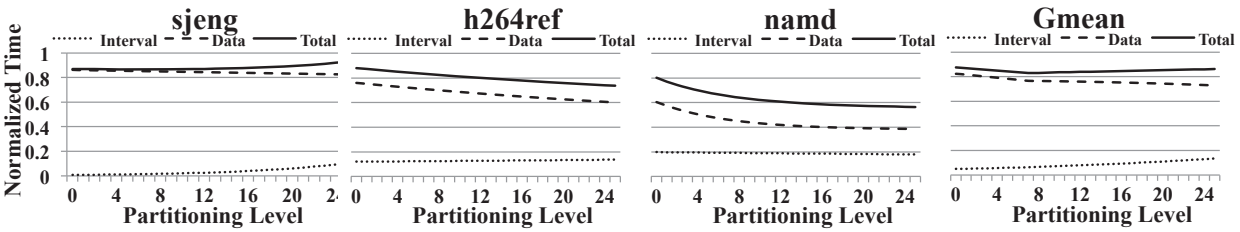

Fig. 9. Normalized access time with different partitioning levels using static partitioning (without timing protection)

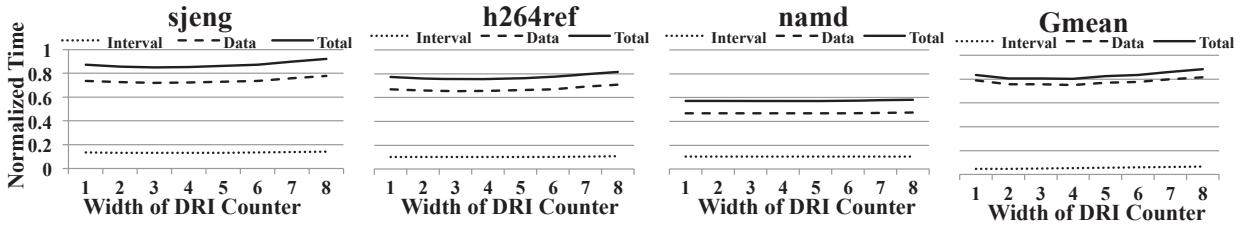

Fig. 10. Normalized access time with different widths of DRI Counter using dynamic partitioning

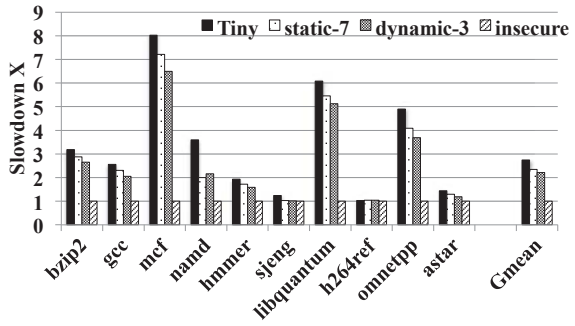

Fig. 11. Slowdown w/o timing protection

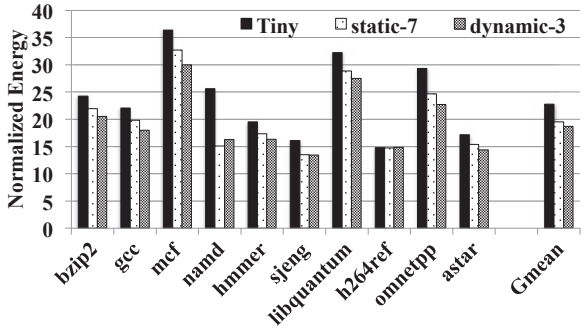

Fig. 12. Energy normalized to the insecure system w/o timing protection

first decreases and then increases with the partitioning level. The minimum total execution time is 83% of Tiny ORAM's execution time when partitioning level is 7.

Figure 10 illustrates the normalized access time when dynamic partitioning is applied. We set the width of DRI Counter from one-bit to eight-bits to investigate the best counter width. We can find that the total execution time first drops and then increases with the increment of counter width. A short-length counter is easily influenced by some noise and cannot catch the feature of LLC miss intervals. For a long-length counter, it may take much time to adapt to the change in LLC miss intervals, which leads to an out-of-date representation of LLC miss intervals. From the results of geometric mean, when the counter's width is set to be 3-bit, the average total execution time can be the minimum, which is 80% of Tiny ORAM's.

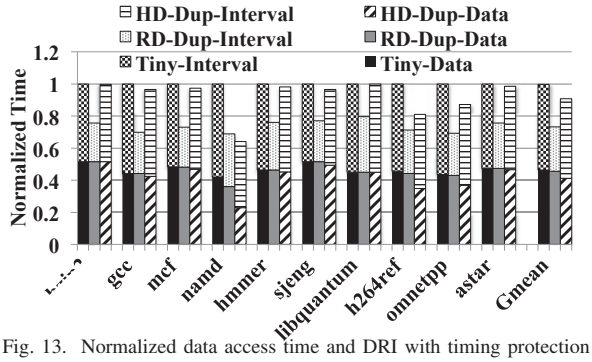

Fig. 13. Normalized data access time and DRI with timing protection

Figure 11 illustrates the slowdown normalized to the insecure system. Static-7 and dynamic-3 represents the static partitioning when partitioning level is 7, and the dynamic partitioning when DRI counter's width is 3-bit. They are chosen as the most efficient schemes among various configurations according to the last section. On average, static-7 and dynamic-3 can achieve 2.35X and 2.21X slowdown in average, which is 85% and 80% w.r.t. Tiny ORAM, respectively. Results of mcf, $libquantum$ and $omnetpp$ show relative high slowdown due to high memory intensity. Using shadow block, speedups of 1.19X, 1.24X and 1.33X are achieved, respectively.

Figure 12 illustrates the energy consumption of memory system normalized to the insecure system. Static-7 and dynamic-3 achieve a reduction of 14% and 18% in energy w.r.t. Tiny ORAM, respectively, since the number of ORAM requests and the total execution time both decrease. Thus, the dynamic power and static power of memory system are saved.

C. Evaluation Results with Timing Protection

In this section, we will evaluate the efficiency of shadow block when timing protection is equipped. We sweep a range of static rates for timing protection in the evaluation of SPEC 2006. Then, we set the static rate of ORAM requests as 800 CPU cycles, which minimizes the average performance overhead of ORAM while maintaining zero-leakage over the timing channel [16]. In other words, a data/dummy ORAM request is launched by the ORAM controller every 800 cycles.

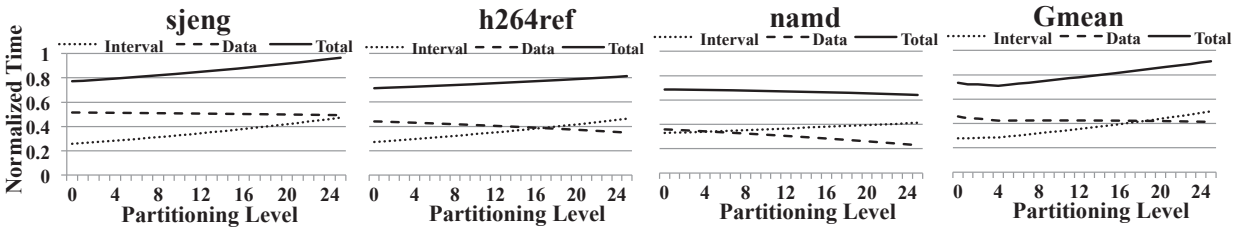

Fig. 14. Normalized access time with different partitioning levels using static partitioning (with timing protection)

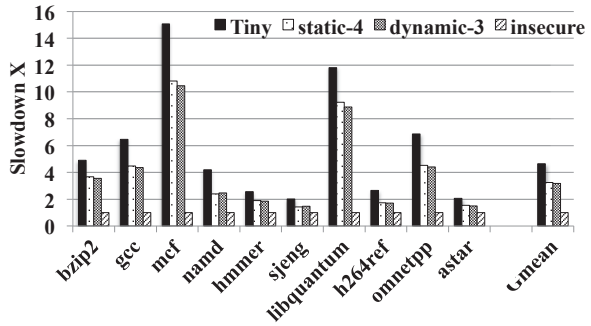

Fig. 15. Slowdown with timing protection

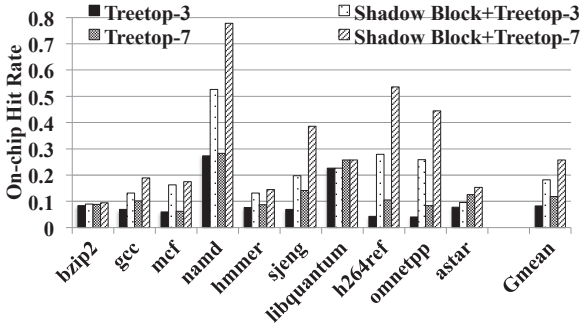

Fig. 16. Hit rate of on-chip stash and treetop caching

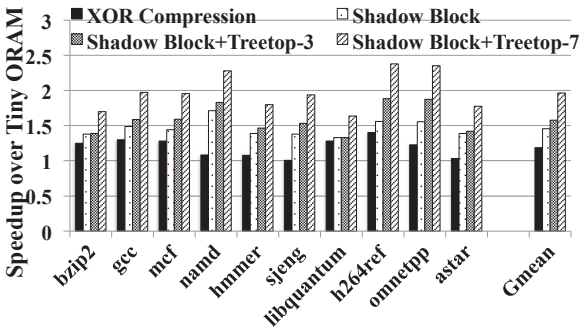

Fig. 17. Comparison with related work

Figure 13 illustrates the normalized execution time when RD-Dup and HD-Dup are adopted, respectively. We can see that the ratio of DRI increases largely due to the injection of dummy requests. Similar as Figure 13, we can see that RD-Dup mainly reduces the DRI and HD-Dup mainly reduce the number of data requests. On average, RD-Dup reduces 48% DRI, 2% data access time and 27% total execution time compared with Tiny ORAM, respectively. By contrast, HD-Dup reduces 7% DRI, 12% data access time and 11% total execution time compared with Tiny ORAM, respectively.

The efficiency of static partitioning and dynamic partitioning are also evaluated. Figure 14 illustrates the normalized access time of $sjeng, h264ref, namd$ and ten workloads's geometric mean after static partitioning is applied. The data access time and DRI show a similar trend as that in Figure 9. Since the ratio of DRI is much larger than that in Figure 9, and thus the partition level should be lowered to facilitate RD-Dup. Experiments show that the best partitioning level is 4, which is less than that in Figure 9. For dynamic partitioning, the trends of data access time and DRI are quite similar to that in Figure 9. And 3-bit is still the best DRI Counter's width to achieve lowest total execution time.

Figure 15 illustrates the slowdown of Tiny ORAM, static-4 and dynamic-3 normalized to the insecure system. Since the energy consumption of memory subsystem is proportional to the number of accesses, the slowdown can also represent the normalized energy consumption of memory system [16]. We can find that both of the static or the dynamic partitioning have a significant speedup over the Tiny ORAM. For static partitioning and dynamic partitioning, an overage reduction of 30% and 32% in execution time is achieved, respectively. Compared to ORAM without timing protection, higher reduction is achieved because dummy ORAM requests are avoided.

D. Comparison with Previous Work

In this section, we present the combinational effect with previous work and the comparison results with XOR compression [34]. For simplicity, we use dynamic-3 as a representative of our design. We choose Treetop caching [15], as one of the state-of-art ORAM optimizations, to illustrate the combinational effect. Since both XOR compression and treetop caching adopt timing protection, we assume timing protection is equipped in this section.

Shadow block can significantly increase the hit rate of on-chip storage (stash, treetop cache). We use two schemes to illustrate: "treetop-3" that caches top 3 levels of ORAM tree (proposed in [15]), and "treetop-7" as a comparison. Figure 16 illustrates the hit rate of treetop caching with/without shadow block technique, respectively. The hit rates of treetop-3 and treetop-7 increase to 2.20X and 2.17X on average with shadow block, respectively. Since shadow block does not introduce on-chip data cache, the increase is mainly caused by the shadow block in the stash or treetop cache, which stores nonce before.

Figure 17 illustrates the speedup over Tiny ORAM when shadow block, XOR compression or combinational optimizations are adopted, respectively. On average, shadow block outperforms XOR compression by 23%. For some benchmarks ($sjeng, namd, astar$), XOR compression shows limited effect while a considerable speedup with shadow block. When shadow block is combined with treetop-3 and treetop-7, performance is further optimized by 8.2% and 23%, respectively.

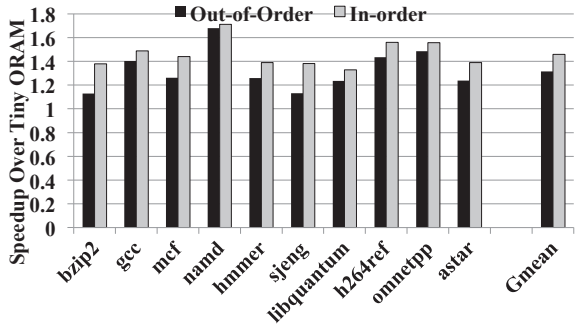

Fig. 18. Speedup with different CPU types

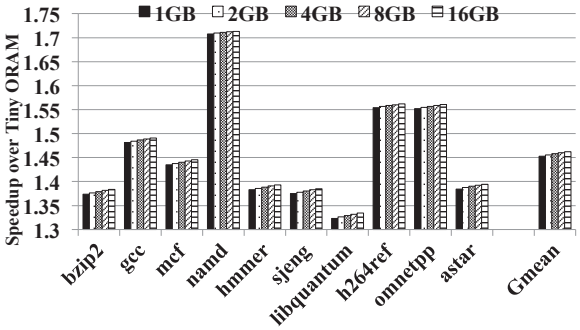

Fig. 19. Speedup with different data ORAM sizes

E. Sensitivity Analysis

In this section, we will evaluate our design under different configurations, including the different CPU types, and different data ORAM sizes. For simplicity, we use dynamic-3 with timing protection as a representative of our design.

Figure 18 illustrates the speedup over Tiny ORAM when CPU types are single-core in-order and quad-core out-of-order, respectively (details in Table I). For the O3 CPU, benchmarks are simply duplicated to ensure that one task on each core. Compared to the speedup when in-order core is adopted, the speedup in O3 CPU reduces. Because of the higher memory intensity in O3 CPU, the DRI is less in O3 CPU than that of in-order CPU. Thus, dummy requests are less likely to occur, which makes advancing data requests (i.e. RD-Dup) less effective (see Figure 2 (d) and (e) for intuitions). However, for both CPU types, HD-Dup is not affected and can reduce the number of data ORAM requests.

Figure 19 illustrates the speedup with different ORAM sizes. The impact of ORAM size changes is slight. And a slight increase of speedup is observed with the increment of ORAM sizes. This is because the smaller the ORAM size, the shorter time the path read takes. Thus, the probability of dummy accesses increases, which is beneficial to the RD-Dup.

VII. Related Work

Since Goldreich and Ostrovsky first proposed Oblivious RAM by [47], [48], enormous follow-up work has been proposed to increase the efficiency of ORAM [10]–[15], [18], [19], [34], [49], [50]. The most related work is about the timing protection of ORAMs [16], the prefetching techniques of ORAMs [17] and XOR compression [31] that utilizes dummy blocks. Besides, many work have been proposed to improve the ORAM efficiency based on novel threat models or novel hardware [51]–[53].

Fletcher *et.al.* propose a ORAM scheme, which can dynamically adjust the rate to launch requests to protect the timing channel of Path ORAM [16]. With a very limited leakage of information, their scheme can save 30% performance overhead compared to a perfect scheme (zero-leakage) with the same power comsumption. In our work, our scheme is the zero-leakage scheme, which is different from this work.

Yu *et al.* propose PrORAM [17] to implement dynamic prefetching in Path ORAM. They observe that ORAM accesses conflict with conventional main memory prefetching. To mitigate the impact, they propose dynamic prefetching, in which multiple adjacent blocks are dynamically constrained to be assigned to the same path. Their results demonstrate that PrORAM can gain a twice performance improvement than previous work on average. Techniques in PrORAM and Shadow block can be combined for further improvement.

Stefanov *et al.* [12] and Devadas *et al.* [31] have proposed XOR compression to reduce the bandwidth overhead of ORAM. With XOR compression, all blocks along a path are first XORed, and then only the XOR result is sent to the processor. However, XOR compression requires computing capability of DRAMs and has limited effect in reducing memory access latency compared to our solution.

Besides the direct optimizations to the ORAM protocol or the ORAM controller, ORAM is further optimized based on new security assumptions or new technologies, such as "trusted memory buffer/logic" or "mixed secure/insecure applications". Aga *et al.* propose to further delegate trust to the memory based on the 3D-stacked new structure of memories, which enables the DRAM with cryptographic computation. Their proposal can significantly reduce the ORAM overhead [53]. Shafiee *et al.* further propose architectural optimizations to the DRAM to mitigate the overhead of Path ORAM, including split the ORAM buckets to parallelize the DRAM accesses and propose secure buffer to further increase the parallelism [52]. Wang *et al.* propose CP-ORAM to schedule ORAM requests with normal request to maximize the server performance [51]. They also propose D-ORAM to leverage buffer-on-board (BoB) of DRAM as the secure delegator to speedup ORAM accesses [54]. These optimizations are orthogonal to our scheme and can be combined with ours.

VIII. Conclusion

In this work, we reveal an important fact that the access order of the intended data block in each ORAM request can impact its performance significantly. If the intended data block is accessed earlier in an ORAM request, the intervals between data ORAM requests can be reduced. Thus, to advance the accesses of intended blocks without compromising security, we propose a novel data duplication technique. The basic idea is to duplicate data blocks in the dummy blocks located closer to the root of an ORAM tree. These dummy blocks with duplicated data are called shadow blocks. With the help of shadow blocks, we introduce two duplication methods to advance accesses of different blocks. Both of them can help improve the ORAM performance and cooperate with each other using an ORAM partitioning technique. Compared to the state-of-the-art ORAM, our design can achieve a considerable reduction in execution time.

REFERENCES

[1] T. Group *et al.*, "Tcg specification architecture overview revision 1.2," 2004.

[2] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, 2000.

[3] 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 17th annual international conference on Supercomputing*. ACM, 2003.

[4] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[5] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872377

[6] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *Computer Architecture, International Symposium on*. IEEE Computer Society, 2005.

[7] X. Zhuang, T. Zhang, and S. Pande, "Hide: an infrastructure for efficiently protecting information leakage on the address bus," in *ACM SIGPLAN Notices*, vol. 39, no. 11. ACM, 2004.

[8] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[9] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[10] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious ram simulation," in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2012.

[11] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013.

[12] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *arXiv preprint arXiv:1106.3652*, 2011.

[13] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.

[14] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas, "Freecursive oram:[nearly] free recursion and integrity verification for position-based oblivious ram," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[15] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "Phantom: Practical oblivious computation in a secure processor," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.

[16] C. W. Fletchery, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014.

[17] X. Yu, S. K. Haider, L. Ren, C. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "Proram: dynamic prefetcher for oblivious ram," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015.

[18] C. W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, and S. Devadas, "Raw path oram: A low-latency, low-area hardware oram controller with integrity verification," IACR Cryptology ePrint Archive, Tech. Rep., 2014.

[19] X. Zhang, G. Sun, C. Zhang, W. Zhang, Y. Liang, T. Wang, Y. Chen, and J. Di, "Fork path: improving efficiency of oram by removing redundant memory accesses," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[20] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007.

[21] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003.

[22] T. S. Lehman, A. D. Hilton, and B. C. Lee, "Poisonivy: Safe speculation for secure memory," in *Proceedings of the 49th International Symposium on Microarchitecture*. ACM, 2016.

[23] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014.

[24] M. Yan, Y. Shalabi, and J. Torrellas, "Replayconfusion: detecting cache-based covert channel attacks using record and replay," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[25] R. Callan, A. Zajic, and M. Prvulovic, "Fase: Finding amplitude-modulated side-channel emanations," in *ACM/IEEE International Symposium on Computer Architecture*, 2015.

[26] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "Eddie: Em-based detection of deviations in program execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture. ACM*, 2017.

[27] F. Liu and R. B. Lee, "Random fill cache architecture," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014.

[28] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016.

[29] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[31] S. Devadas, M. V. Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, *Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM*. Springer Berlin Heidelberg, 2016.

[32] Y. Wang, B. Wu, and G. E. Suh, "Secure dynamic memory scheduling against timing channel attacks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017.

[33] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014.

[34] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Ring oram: Closing the gap between small and large client storage oblivious ram." *IACR Cryptology ePrint Archive*, vol. 2014, 2014.

[35] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious ram simulation," in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, 2012.

[36] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o((logn)3) worst-case cost." *Lecture Notes in Computer Science*, vol. 2011, 2011.

[37] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi, "Scoram: Oblivious ram for secure computation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660365

[38] S. Sechrest, C.-C. Lee, and T. Mudge, "The role of adaptivity in two-level adaptive branch prediction," in *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE Computer Society Press, 1995.

[39] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014.

[40] A. Ferraiuolo, Y. Wang, D. Zhang, A. C. Myers, and G. E. Suh, "Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016.

[41] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1. IEEE, 1999.

[42] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[43] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, 2011.

[44] H. Bhatnagar, *Advanced ASIC Chip Synthesis: Using Synopsys® Design CompilerTM Physical CompilerTM and PrimeTime®*. Springer Science & Business Media, 2007.

[45] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," Technical Report 2001/2, Compaq Computer Corporation, Tech. Rep., 2001.

[46] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, 2006.

[47] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, 1996.

[48] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987.

[49] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o ((logn) 3) worst-case cost," in *Advances in Cryptology–ASIACRYPT 2011*. Springer, 2011, pp. 197–214.

[50] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in) security of hash-based oblivious ram and a new balancing scheme," in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2012.

[51] R. Wang, Y. Zhang, and J. Yang, "Cooperative path-oram for effective memory bandwidth sharing in server settings," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017.

[52] A. Shafiee, R. Balasubramonian, M. Tiwari, and F. Li, "Secure dimm: Moving oram primitives closer to memory," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018.

[53] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[54] R. Wang, Y. Zhang, and J. Yang, "D-oram: Path-oram delegation for low execution interference on cloud servers with untrusted memory," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018.