Fork Path: Improving Efficiency of ORAM by Removing Redundant Memory Accesses

Xian Zhang[†]
zhang.xian@pku.edu.cn

Guangyu Sun[†,‡]
gsun@pku.edu.cn

Chao Zhang[†]
zhang.chao@pku.edu.cn

Weiqi Zhang[†]
zhangweiqi@pku.edu.cn

Yun Liang[†,‡]
ericlyun@pku.edu.cn

Tao Wang[†,‡]
wangtao@pku.edu.cn

Yiran Chen[¶]
yic52@pitt.edu

Jia Di[§]
jdi@uark.edu

[†]Center for Energy-Efficient Computing and Applications, Peking University, Beijing 100871, China
[‡]Collaborative Innovation Center of High Performance Computing, NUDT, Changsha 410073, China
[¶]Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh PA 15261, USA
[§]Computer Science and Computer Engineering Department, University of Arkansas, Fayetteville AR 72701, USA

ABSTRACT

Oblivious RAM (ORAM) is a cryptographic primitive that can prevent information leakage in the access trace to untrusted external memory. It has become an important component in modern secure processors. However, the major obstacle of adopting an ORAM design is the significantly induced overhead in memory accesses. Recently, Path ORAM has attracted attentions from researchers because of its simplicity in algorithms and efficiency in reducing memory access overhead. However, we observe that there exist a lot of redundant memory accesses during the process of ORAM requests. Moreover, we further argue that these redundant memory accesses can be removed without harming security of ORAM. Based on this observation, we propose a novel Fork Path ORAM scheme. By leveraging three optimization techniques, namely, path merging, ORAM request scheduling, and merging-aware caching, Fork Path ORAM can efficiently remove these redundant memory accesses. Based on this scheme, a detailed ORAM controller architecture is proposed and comprehensive experiments are performed. Compared to traditional Path ORAM approaches, our Fork Path ORAM can reduce overall performance overhead and power consumption of memory system by 58% and 38%, respectively, with negligible design overhead.

MICRO-48, December 05-09, 2015, Waikiki, HI, USA
©2015 ACM. ISBN 978-1-4503-4034-2/15/12 ...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830787.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Miscellaneous; K.6 [**Management of Computing and Information Systems**]: Security and Protection.

Keywords

Oblivious RAM, Access Merging, Request Scheduling

1. INTRODUCTION

Following the fast growth of consumer electronics and cloud computing industry, the demand for data security and privacy protection keeps increasing. Recently, secure processors, which can co-operate with software countermeasures to offer holistic protection [1, 2], have been widely proposed to enhance hardware security. Since traditional secure processor designs focus on the security of data content, there has been many research works on data encryption to protect data stored in external memory [3, 4, 5]. However, recent research has pointed out that even with data encryption, the data access pattern can still leak considerable sensitive information [6, 7, 8]. To overcome this problem, oblivious RAM (ORAM) is extensively investigated lately.

ORAM is a cryptographic primitive that can conceal access patterns to memory so that the information leakage in a program's memory access trace can be eliminated [9, 10, 11]. The basic idea is that ORAM maintains an encrypted and shuffled form for all data stored in memory. For each memory access, data are re-encrypted and reshuffled. In ORAM, a memory access pattern is computationally indistinguishable from the others with the same length [12, 13]. Ever since ORAM was first proposed in 1987 [14, 15], it has attracted more and more attentions in the security community, including some very recent practices [9, 16, 17, 10, 11]

As mentioned in many previous works, the main limitation of ORAM technique is its large access overhead in memory accesses. Compared to the unprotected baseline, ORAM induces $10\times-100\times$ more memory accesses [12, 18, 19] leading to significant virtual increase in memory access latency. It could result in up to $10\times$ system-level performance degradation, especially for those memory intensive applications [13, 18, 12]. Since many secure processors have adopted chip-multi-processor (CMP) and out-of-order pipelining architectures to improve performance and compatibility with commercial insecure processor [20, 21, 22, 23], the limited external memory bandwidth has already become a bottleneck. ORAM, thus, drastically aggravates this already-severe issue.

Recently, an ORAM scheme called Path ORAM is proposed [10] with high algorithm efficiency and simplicity. In Path ORAM, the external memory is logically structured as a binary tree. The processor accesses the memory with a random path descending from one leaf to the root of the binary tree. Several follow-up work and techniques have been proposed to further reduce the access overhead of Path ORAM [13, 18, 12]. Thus, current variances of Path ORAM are considered as the state-of-art and most efficient approaches. However, for many memory-intensive applications, the overhead of Path ORAM is still too high for practical usage.

We find that it is still possible to continue improving memory access efficiency of Path ORAM by observing the substantial redundant memory accesses during data process. Simply speaking, each ORAM request is processed separately by traversing a complete ORAM path in the binary tree mentioned above. In fact, from the perspective of a sequence of consecutive ORAM requests, a lot of memory requests are redundant and can be removed without harming security of ORAM design.

Based on this observation, we propose a **Fork Path ORAM** scheme in this work to remove those redundant memory accesses efficiently and safely. The major contributions of this work can be summarized as:

1. Instead of operating each ORAM request independently, we process ORAM requests by considering their previous request and following request. Then, we argue that some memory accesses can be removed without leaking information;
2. We propose a path merging technique to remove those redundant memory accesses;
3. We introduce a request scheduling technique to improve the efficiency of path merging;
4. After applying path merging, we present a new merging-aware caching scheme to further decrease the number of memory accesses per ORAM access;
5. A detailed architecture of ORAM controller is proposed and comprehensive evaluation and comparison are presented.

The rest of this paper is organized as follows: Section 2 introduces the threat model and basic knowledge of ORAM. In addition, a state-of-art Path ORAM scheme is described as the baseline of this work; Section 3 introduces Fork Path ORAM, which is composed of three components, i.e., path merging, request scheduling and merging aware caching; Section 4 presents the detailed architecture of Fork Path ORAM controller; Section 5 gives a comprehensive evaluation of our design and compares it with the baseline Path ORAM in terms of performance, energy, and design overhead; Section 6 summarizes the related works, followed by a conclusion section.

2. BACKGROUND

In order to understand the design target of ORAM, in this section, the threat model needs to be presented first. Then, the basic idea of ORAM is introduced to explain how the memory is protected from the threats. Lastly, a state-of-art design, Path ORAM, is elaborated to illustrate the memory access flow under the current ORAM protection. Observations are concluded to motivate the optimization techniques proposed in this work.

2.1 Threat Model

The threat model used here is similar to those proposed in previous works [12, 18]. User's private programs are running on a processor interacting with an external memory. As assumed in previous work [12, 18, 13], the processor is trusted. It means that all data inside this secure processor are invisible to the outer adversary. However, the external memory is exposed to the adversary. In other words, the external memory is untrusted; the adversary is capable of capturing all the information in the memory, including data, addresses on the bus and their corresponding timing information.

Existing secure processors have provided encryption to protect data contents [4, 1, 2, 24, 5]. However, based on the access patterns on memory bus, the adversary can still learn sensitive information of the programs, such as the encryption type or even secret keys [6, 7, 8]. To overcome this problem, Oblivious RAM (ORAM) is proposed to prevent the memory access pattern from being detected.

2.2 ORAM Basics

ORAM is a cryptographic primitive that can completely hide the memory access patterns by transforming the memory request sequence into a random request sequence(i.e. ORAM requests). As proved in previous work, an ORAM design that satisfies the following rule is considered to be secure [13, 10, 11]:

For any two data request sequences $\overrightarrow{a}$ and $\overrightarrow{a}'$, which are composed of $(address, operation, writedata)$ tuples that are compatible to a standard RAM interface, their resulting sequences $ORAM(\overrightarrow{a})$ and $ORAM(\overrightarrow{a}')$ are computationally indistinguishable if these two resulting sequences have the same length.

The $\overrightarrow{a}$ represents the sequence of memory load/store requests from the program. Since on-chip caches (which is trusted) are normally employed in a secure processor, $ORAM(\overrightarrow{a})$ normally refers to those memory requests caused by misses of last level cache (LLC). As mentioned in previous works, the length of $ORAM(\overrightarrow{a})$ can indicate the number of cache hits [12, 13]. Thus, information leaks logarithmically with the increasing length of $ORAM(\overrightarrow{a})$. However, a nonstop stream of the accesses to the external memory can be used to prevent this leakage. In other words, requests will be issued at a data-independent time no matter whether there are LLC misses or not[13, 25].

Note that the core purpose of ORAM is to protect privacy of memory access patterns (i.e., the address sequence and its timing channel [13, 25]). ORAM can also cooperate with other countermeasures against other attacks such as active attacks[5, 26, 24], EM-attacks[27], convert channel attacks[28], and cache side channel attacks[29], etc. For example, the integrity checking (e.g., Merkel Tree) can be combined with ORAM to counteract active attacks[18, 12]. These work are considered as orthogonal to ORAM designs and out of the scope of this paper.

2.3 Path ORAM

We use a state-of-art ORAM design, namely, Path ORAM as an example to illustrate the memory access flow after an LLC miss in ORAM. Path ORAM is an efficient and algorithmically simple Oblivious RAM, which has been proposed to be a component in a secure processor[18, 12, 13, 30]. Recent prototypes have proved that Path ORAM is one of the most promising and efficient ORAM implementations [12, 13]. An overview of Path ORAM architecture can be found in Figure 1. It includes two components: (1) a trusted on-chip ORAM controller (upper half) and (2) an untrusted external memory (lower half).

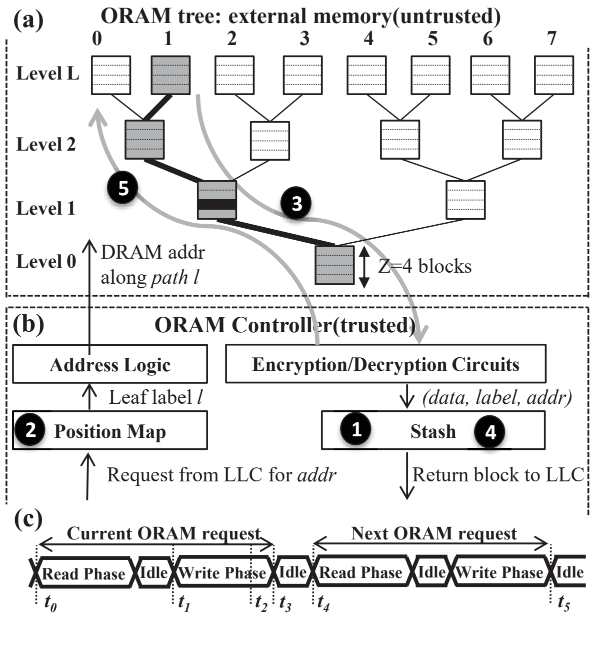

Figure 1: An illustration of Path ORAM architecture[12] that consists of (a) an ORAM tree and (b) an ORAM controller ($L = 3$ and $Z = 4$); (c) the timing diagram of memory bus.

The external memory is logically organized as a binary tree, which is called **ORAM tree** [12, 13]. As shown in Figure 1(a), the binary tree has $L + 1$ levels, ranging from level 0 (root) to level L (leaf). Each node of the tree contains one bucket, which holds a fixed number (denoted as Z) of slots to store memory blocks. In each bucket, the number of *data blocks*, which store valid data of the program, varies between $0 \sim Z$. The rest of a bucket is filled with *dummy blocks*. Both data blocks and dummy blocks are encrypted with probabilistic encryption (e.g., counter-mode [4, 18]). It means that any two blocks are indistinguishable even their plain data are the same, regardless of being dummy or real blocks. Each leaf node of the ORAM tree is assigned a unique label in order. And path-l is defined as the path descending from the leaf with label l to the root. For example, in Figure 1(a), path-1 is highlighted in grey.

The ORAM controller consists of a *stash*, a *position map*, and some control logics as shown in Figure1(b). The stash is an on-chip memory component, which can temporarily hold a small number (e.g. 200 data blocks [12, 18]) of data blocks. The position map is a lookup table recording the run-time mapping relationship between data blocks and leaf labels. During program execution, each data block is randomly mapped to a leaf label at runtime. Path ORAM design holds the following **invariant**[10, 18]:*a data block mapped to leaf label l must be either in the stash or path l.* Data blocks are stored together with their leaf labels and program addresses (a.k.a. query address from CPU), no matter in the external memory or in the stash.

For every memory request denoted as $(addr, op, data)$, it is transformed into an ORAM request in following steps [13], which are also illustrated in Figure 1:

- **Step 1** Stash is searched for a data block with $addr$. If the data block is in the stash, it is returned to LLC immediately.
- **Step 2** If a stash miss happens, the leaf label (l) of the data block being searched is identified from the position map by indexing with $addr$. Then, the data block is re-mapped to a new leaf label l' and the position map is updated.
- **Step 3** All blocks along path-l are loaded from external memory and stored into stash after decryption. Among these blocks, only those actually required by the processor are forwarded to LLC.
- **Step 4** Since the data block has been re-mapped to a new label l' in Step 2, its label is updated to l' in stash. Thus, only the block in stash is valid. The data block in memory becomes out-of-date.
- **Step 5** Path l needs to be re-filled (overwritten) with new blocks. The basic rule is to re-fill the path with data blocks in stash as many as possible [13, 10]. Note that data blocks written back to memory are evicted from stash to keep the invariant. If there still exist free slots in the path, dummy blocks are inserted.

To summarize, the Path ORAM maps the original memory access sequence into a random "leaf label sequence". The security of Path ORAM relies on the fact that every label in the sequence is random and independent to the previous labels [10]. Thus, no address information leaks if the label sequence is obtained. The ORAM's timing channel can also be protected when each ORAM request is launched at a data-independent time. An example is shown in Figure 1(c). It is easy to find that there is a fixed interval (idle phase) between every adjacent read and write phases. Thus, even when there are no data requests from LLC misses, **dummy ORAM requests** should be still launched [25, 13].

It is worth mentioning that Path ORAM can encounter a deadlock when all the buckets along the refilled path are full, which contributes to stash overflow. To mitigate the possibility of stash overflow, proper configurations of DRAM utilization, stash size C and bucket size Z should be set [18]. For example, when the utilization of a 8GB DRAM is 50% while $C \geq 200$ and $Z \geq 4$, the possibility is negligible[10, 13, 12].

Due to the limited on-chip storage, if the block size of the memory is relatively small (e.g. 64B or 128B), it is difficult to maintain the entire position map inside a secure processor. For example, for a 4GB data working set and 64B memory block size, the number of data blocks is $N = 64M$ and the size of position map is $192MB$. To solve this problem, hierarchical path ORAM is proposed[10, 18, 12]. The basic idea is to store the position map in external memory and also protect it with ORAM. Similar to the protection of the data, the memory storage for the position map is also organized as a new ORAM tree. A new position map is then generated for this new ORAM tree. In order to differentiate it from the original data ORAM, this new ORAM for position map is called $ORAM_1$, as shown in Figure 2.

With this hierarchical architecture, a data request from LLC is completed with accesses to two ORAMs. First, $ORAM_1$ is accessed to retrieve the label of data block from the position map protected. Then, the data ORAM is accessed with this label to retrieve data blocks. The access flow of this hierarchical ORAM is illustrated in Figure 2. For the above example, $7.5MB$ memory is required to store position map of $ORAM_1$. If this is still too large to be stored on-chip, extra levels can be added to this ORAM hierarchy recursively until the position map of the last level ORAM can be stored in the secure processor, which is also illustrated in Figure 2.

To improve the efficiency of using hierarchical Path ORAM, an unified program address space is allocated for all ORAM trees in the hierarchy, as shown in Figure 2(b) [12]. This implementation can avoid a considerable amount of overhead in timing channel protection to hide the position map hit/miss. Thus, from the view outside the processor, ORAM requests of accessing different levels of the ORAM hierarchy are indistinguishable. In other words, the hierarchical ORAM behaves the same as the basic Path ORAM. In addition, there is only one stash for the unified ORAM tree. The only difference is that one program address request may be transformed into multiple program address requests.

In the rest of this paper, the unified hierarchical Path ORAM is used as our baseline for discussions and denoted as Path ORAM for simplicity.

3. FORK PATH ORAM SCHEME

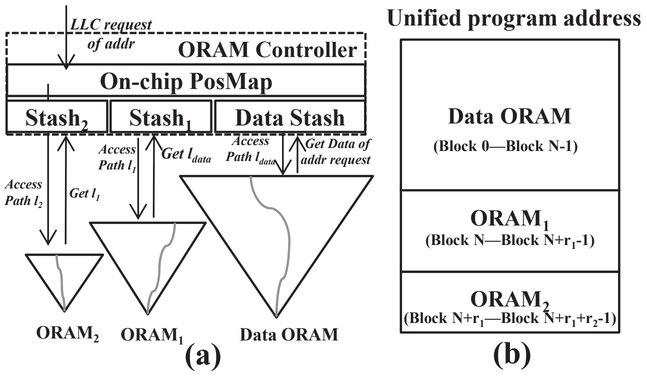

Figure 2: (a)An illustration of hierarchical Path ORAM architecture(two-level) and memory access flow (b) Hierarchical Path ORAM in a unified program address space

In this section, we will use a simple example to illustrate the redundancy in the memory accesses in traditional Path ORAM. We then propose path merging to remove such redundancy, followed by ORAM request scheduling and merging-aware caching techniques to further enhance the efficiency of path merging .

3.1 Motivation

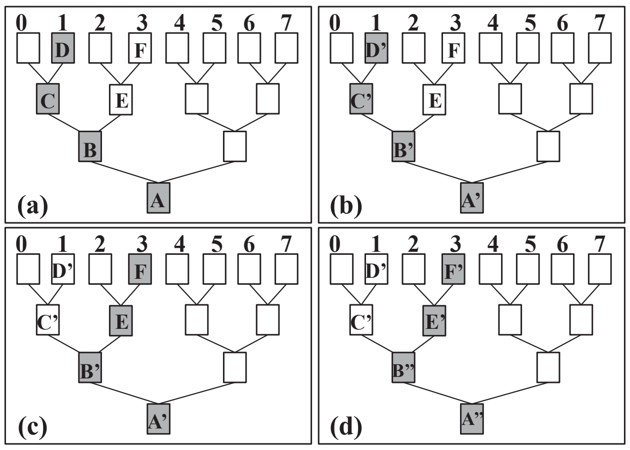

Figure 3: Read/write phases for two adjacent requests accessing path-1 and path-3:(a) Read phase of path-1,(b) Write phase of path-1, (c) Read phase of path-3, (d) Write phase of path-3

As shown in Figure 3, each block in the ORAM tree represents a data bucket and the letter in a bucket represents data stored in it. In this example, two ORAM paths with leaf label 1 and label 3 are accessed consecutively. Based on the flow introduced in the last section, the access sequence to memory used in the traditional Path ORAM design is depicted in Figure 3(a)∼(d): At first, all data in buckets along path-1 (A, B, C, D) are decrypted and loaded into the stash. Then, path-1 is filled with write-back data (A', B', C', D'). Similarly, for the access to path-3, old data along the path (A',

B', E, F) are loaded into the stash followed by the write-back process with new data (A'', B'', E', F').

Considering the overlapped part of path-1 and path-3, it is easy to find that data A' and B' are written to the external memory and then loaded back into secure processor intactly. Thus, an important observation can be summarized as follows.

Observation: Memory operations of writing and reading data in the overlapped region of two consecutive ORAM requests are considered to be redundant. This is public information that visible to anyone including the adversary.

Hence, if these redundant accesses can be removed, the memory access efficiency of Path ORAM will be improved without leaking any information to the adversary. This fact motivates the path merging technique introduced in the next subsection.

3.2 Path Merging

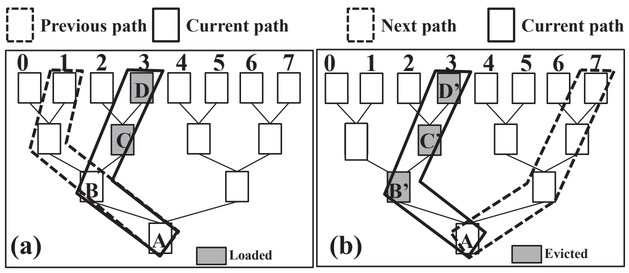

Figure 4: Illustration of path merging: (a) Read phase of current path (b) Write phase of current path

Based on the fact aforementioned in Section 3.1, we proposed path merging technique. The basic idea of path merging is to "merge" two adjacent ORAM requests and avoid redundant accesses to their overlapped path. Using the example in Figure 4, the basic ORAM request process flow can be modified as follows:

- **Step 0** For the first ORAM request after initialization of the entire system, all blocks along the path are loaded from external memory and stored into stash after decryption. Among these blocks, only those are actually requested by processor are forwarded to LLC;
- **Step 1-2** are kept unchanged as in Section 2.3;
- **Step 3** Only those buckets along a certain part of the path, i.e. NOT overlapped with the path of the previous ORAM request, are loaded from the external memory and stored into the stash after decryption. Among these blocks, only those are actually requested by the processor are forwarded to the LLC. For example, in Figure 4(a), only C and D are loaded from the external memory;
- **Step 4** is kept unchanged as in Section 2.3;
- **Step 5** When current path l needs to be refilled, if there exists a pending (next) ORAM request, only those buckets along the part NOT overlapped with the path of the next ORAM request are refilled. As shown in Figure 4 (b), since the next ORAM request will access path-7, only B', C', and D' are refilled.
- **Step 6** When the current path l needs to be refilled, if there is no pending (next) ORAM request, a dummy ORAM request will be inserted. Then, blocks are refilled as in Step 5.

As a summary, by modifying read and write (refill) steps, path merging avoid redundant accesses to the overlapped part of two consecutive ORAM requests. As depicted in Figure 4, after adopting path merging, these blocks are accessed in the shape of a **fork path**. Since redundant dummy requests may be inserted before the refill process, extra requests may be induced w.r.t. traditional Path ORAM design. In order to mitigate this problem, a dummy request replacing technique is introduced in the next subsection.

3.3 Dummy Label Replacing

After path merging is used, the process of an ORAM request will rely on the previous and the subsequent ORAM requests. Thus, a dummy ORAM request is inserted when there are no pending requests to be merged. In other words, if the memory request intensity from LLC is low, more dummy requests may be induced compared to the traditional Path ORAM design. This situation becomes common in the case where the secure processor is in-order and single-core, especially when hierarchical Path ORAM is employed. Thus, the benefits of using path merging may be offset by these extra dummy ORAM requests. While this problem can be mitigated by simply extending the idle interval between read and write phases, the latency of ORAM requests will be prolonged. In this work, we propose a dummy request replacing technique to solve this issue.

In fact, after a dummy ORAM request is inserted for path merging, there is still a chance that it can be replaced by the later incoming real data request without being noticed from outside the secure processor. Rationale relies on the fact that a refill (write) process starts from the leaf and descends toward the root in a path. The dummy request is not revealed until the refill process of the current ORAM request is completed. Thus, if the dummy request is replaced before this point, it is invisible to the adversary outside the secure processor. An example depicted in Figure 5 explains how a dummy ORAM request can be replaced.

As shown in the figure, there are three ORAM requests corresponding to three ORAM paths: (1) path-0, the current ORAM path being processed, (2) path-7, the dummy path inserted for merging, and (3) path-3, the data path that arrives after insertion of the dummy path. As shown in the figure, the current being processed is in its refill step. Those gray blocks on the path represent the buckets that have been updated, and the rest blank blocks are those to be updated. Whether the dummy ORAM request can be replaced by the later real data ORAM request depends on the completion status of refill process when the real data ORAM request arrives, including three different cases as below:

As shown in Figure 5(a), the current ORAM path and the dummy ORAM path cross at the bucket-A.

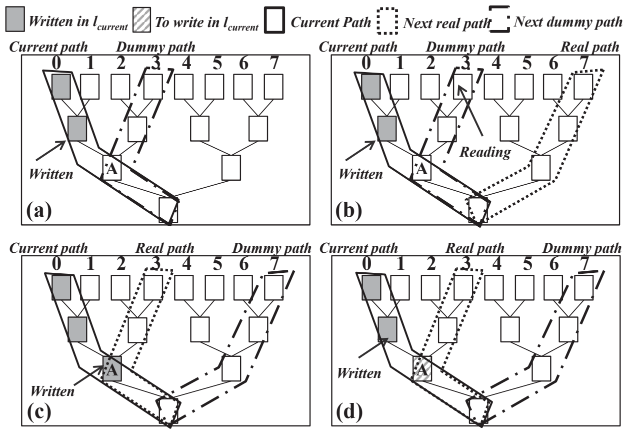

Figure 5: Illustration of dummy request replacing: (a) initial state, (b) case-1, (c) case-2, (d) case-3

- **Case-1** If the data ORAM request arrives after the refill process has completed (i.e., buckets above bucket-A have been updated), the dummy request cannot be replaced.
- **Case-2** When the data ORAM request arrives, the current refill process is not yet finished. However, the bucket on the crossing point of the current ORAM path and the data ORAM path has been updated already. The dummy request cannot be replaced.
- **Case-3** For the rest cases, the dummy ORAM request can be replaced by the data ORAM request.

The proposed dummy label replacing technique can efficiently reduce the number of extra dummy requests, as illustrated in Figure 1(c). We use t_2 to denote the time when the cross point of the current path and the dummy path is written. If the next real request appears during $t_1 - t_2$, dummy label replacing can replace the dummy request inserted at t_1 instead of waiting for the completion of the dummy request. Consequently, if the next request appears during $t_0 - t_2$, there will be no extra dummy requests, similar to the original ORAM.

It is easy to observe that the efficiency of both path merging and dummy label replacing rely on the overlap degree of two consecutive ORAM requests. In the next subsection, an ORAM request scheduling technique is proposed to enhance the overlap degree at runtime.

3.4 ORAM Request Scheduling

The basic path merging method only focuses on the two consecutive ORAM requests. In fact, when the memory access intensity is high, there may exist multiple pending ORAM requests. This is quite common in secure processors with multi-core and/or using out-of-order pipelines. Therefore, it is possible to reschedule the processing order of these requests to improve the efficiency of path merging.

The modified ORAM process flow can be described as follows: Whenever there is a memory request from LLC, the request is transformed to ORAM requests as

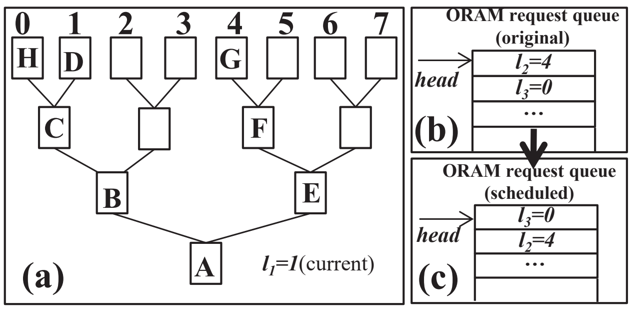

Figure 6: Illustration of ORAM request scheduling: (a) ORAM tree (b) requests before scheduling (c) requests after scheduling

soon as possible and inserted into a queue that recording path labels of these ORAM requests, as shown in Figure 6. Note that for the PosMap miss in hierarchical Path ORAM, there will still be an ORAM request to be inserted into the queue. When the current ORAM path is being loaded, among all pending ORAM requests in the queue, the request that has the highest overlap degree is selected and scheduled as the next ORAM request for path merging.

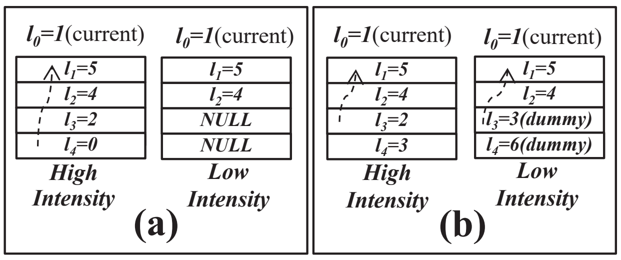

Figure 7: (a) Scheduling among a variable number of pending requests will leak information (b) Dummy labels should be inserted if the queue is not full with data ORAM request.

Figure 6 depicts an example of the detailed ORAM request scheduling flow. The current ORAM request being processed is path-1 and the pending requests will access path-4 and path-0. In this case, ORAM request visiting path-0 should be scheduled before the one visiting path-4. It is because path-0 has more overlapped part with path-1, which is being processed, as also illustrated in Figure 6. Note that rescheduling memory requests may cause data hazards and fairness issues, which have been addressed in some previous works [13, 10]. These problems are also addressed and solved in the Fork Path ORAM architecture, which is introduced in Section 4.

On average, the more ORAM requests pending in the queue, the higher efficiency path merging with scheduling can achieve. However, using an adaptive scheduling depending on the number of the pending data ORAM requests in the queue may raise some security concerns. As shown in Figure 7(a), if every time we take all pending ORAM requests into account to perform scheduling, the degree of path overlapping will reflect the intensity of LLC requests. To prevent this leakage, every time we just ensure the queue is full with ORAM requests. If there are no enough data ORAM requests, dummy requests are inserted. Note that, when we perform scheduling among the requests in the queue, the real request has a higher priority to be launched than that of the dummy request if their overlap degrees with the current path are the same.

Algorithm 1: Label insertion.

```
while $time$ ++ do
    if $current$ is finish then
        current = pending;
        pending = queue top ;
        pop queue top;
    else
    end
    if there is a new request then
        if $dist(current, incoming) <$
        $dist(current, pending)$ and pending is not
        merged then
            swap the pending and incoming
            requests;
        else
        end
        replace the first dummy request with
        incoming request;
        sort the queue by the overlap degree;
    else
    end
    if the queue is not full and have no dummy
    request then
        Insert a dummy request to end of the queue;
    else
    end
end
```

Dummy label replacing can still function during the insertion to the label queue. For a dummy request in the queue but not revealed to the outside, it can be arbitrarily replaced by a real request newly arrived, which is described in Algorithm 1. It is worth mentioning that after replacing, the remaining dummy request still has a chance to be the next launched request if it has the highest overlap degree with the current path.

3.5 Merging-aware Caching

As pointed out by previous studies [13], on-chip data caching can also help reducing the overhead of ORAM. An on-chip memory in ORAM controller is dedicated to cache frequently accessed data blocks. It has demonstrated that treetop caching is an extremely efficient policy. It means that, statistically, data blocks at lower levels (i.e., closer to the root) of the ORAM tree are more frequently accessed than that at the higher levels of the tree. This is true in traditional Path ORAM scheme because each path is traversed completely. Thus, for a fixed size on-chip memory, it is normally filled with data closer to the root, as illustrated in Figure 8(a).

However, after applying path merging and ORAM request scheduling, the simple treetop caching becomes less efficient. In fact, treetop caching may not be effective when the size of the on-chip memory is not large enough. The reason is easy to understand: After applying path merging, every two consecutive paths are merged into a fork style. Only data blocks on the tines of the fork (paths) will be accessed and those data blocks on the handle of the fork (paths) have already been cached in the stash. Statistically, if the average overlapped path length is assumed as $len_{overlap}$, it is almost useless to cache data in the levels lower than $len_{overlap}$ with the simple treetop caching policy.

Hence, in order to recover the efficiency of on-chip caching, the blocks in the first $len_{overlap}$ levels will bypass the cache. Only the data blocks located in the higher level than $len_{overlap}$ are cached. To differentiate from traditional treetop caching policy, this scheme is called merging-aware caching policy, which is depicted in Figure 8(b). LRU replacement policy is adopted in our merging-aware caching scheme and its indexing is changed as follows: Blocks in the level-m_1 ($m_1 = len_{overlap} + 1$) to level-m_2 are cached where m_2 is determined by the cache size. Each level $r (m_1 \le r \le m_2)$ are allocated with 2^{r-m_1+1} blocks in the cache.

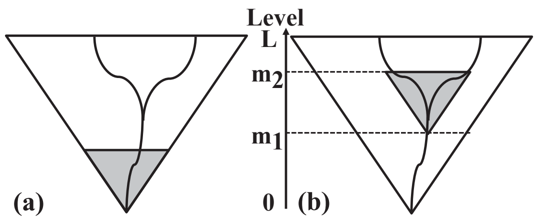

Figure 8: (a) Treetop caching v.s (b) Merging Aware Caching

Here we use a normal cache to implement the address cache. For those levels allocated with blocks more than the number of cache ways, multiple sets will be used to hold those blocks. Every evicted block in these levels from the stash will be inserted to the correspondent set, which is only determined by the logical address of the block(more details will be presented in Section4). A LRU replacement policy is used in our address cache. Note that, since the write phase or the read phase of a Path ORAM starts from the leaf and moves toward the root, a father node are always newer than its son nodes.

For a block at $addr$, we use $level - x$ to denote its level and it is the y-th block at that level from the left. Obviously x and y are determined only by $addr$. If x is not within the range of $[m_1, m_2]$, it is not in the cache. Otherwise, the set number can be calculated as follows:

$$Set\_number = \frac{(2^{x-m_1} - 2) * Z}{cache\_ways} + \frac{(y\%2^{x-m_1+1}) * Z}{cache\_ways} \quad (1)$$

The first item represents the number of sets allocated for the buckets at level-m_1 to level-y. The second item represents the number of sets allocated for the buckets at the same level left to $addr$. Z is the bucket size.

3.6 Security Proof

The security of Path ORAM relies on the independence and randomness of the label sequence, which can hide the original memory access pattern [10]. In path merging, our new modification is **only** based on the label sequence itself, which is sooner or later revealed to the public and leaks no information about the access pattern. In other words, our modification is completely based on the public information. Hence, the ORAM with path merging has the same security strength with the original ORAM.

In addition, path merging will not increase the probability of stash overflow. In the original design, the overlapped part of the old path are first written to the main memory and then loaded up with the non-overlapped part of the new path. When path merging is applied, the overlapped part of the old path will not be written back and then only the non-overlapped part of the new path will be accessed. Obviously, the block numbers of the stash in these two situations are completely the same. Therefore, our merging scheme does not change the possibility of stash overflow.

Similarly, we can prove the security of label scheduling since the scheduling is also based on the label sequence. Also, label scheduling will not change the probability of stash overflow. We can consider label scheduling as the reordering of address requests from LLC. Since the possibility of stash overflow is only related to the level of the ORAM tree and size of the stash [13], for the permutation of address requests, the possibility of stash overflow keeps the same regardless of stash hit or miss. In addition, we remark that as long as we keep the scheduling queue full, the dummy label replacing will leak no information about the LLC intensity. At last, since the security of treetop caching is proved in [13] and the mechanism of our caching scheme is quite similar, the security of our caching scheme can be proved in the same way.

4. FORK PATH ORAM ARCHITECTURE

In order to support Fork Path ORAM, the traditional ORAM controller [18, 12] needs to be modified, as shown in Figure 9. Besides those existing exponents (i.e., a stash and a position map), two request queues and a set-associative cache are added. The basic functions of these components are explained as below:

The first queue is called "address queue" and used to buffer incoming real memory requests from LLC. It stores the program address (PA) of each memory request. An extra bit (R) is added to identify whether the data in the entry is ready. The requests in address queue are sent to position map in order and their corresponding ORAM path labels are output to the second queue, which is called "label queue". As aforementioned, these labels represent the pending ORAM requests to be processed by path merging. Obviously, ORAM request scheduling is also performed in this label queue.

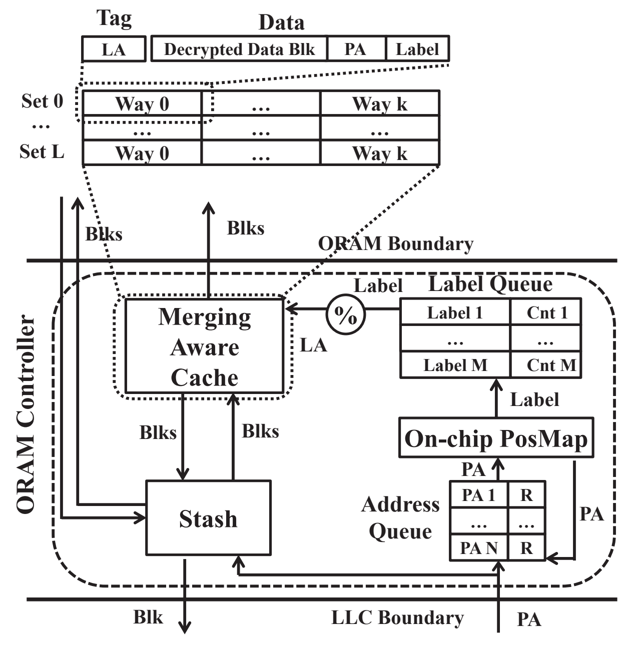

Figure 9: Architecture of the ORAM controller.

Since request scheduling may change the processing order of ORAM requests in the label queue, data hazard problems must be avoided. In fact, this problem should be solved in the address queue to avoid information leakage. There are four possible scenarios that need to be protected by applying proper constraints to the address queue:

- **Read-before-Read**. If two requests read the same address in memory, no specific action is needed.
- **Read-before-Write**. If a read request is followed by a write request to the same address in memory, the write request cannot be sent to the position map (pending in the address queue) until the read request is completed (data ready).
- **Write-before-Read**. If a write request is followed by a read request to the same address in memory, the read request is returned directly with data forwarding.
- **Write-before-Write**. If a write request is followed by another write request to the same address in memory, the first write request is canceled.

By applying these constraints, all requests output from address queue can be scheduled without causing data hazards. After these requests are transferred into ORAM requests, they are inserted into the label queue to achieve the scheduling. Note that each entry of the queue has a counter (Cnt) to remember the "age" of each request. When the counter value of a label (dummy or real) reaches a threshold, this request needs to be prompted to the head of the queue to avoid starvation.

The merging-aware cache only holds the decrypted data that will be written back to memory (e.g. data evicted from stash). Note that data in the cache can also be prompted back to stash if they are hit by ORAM requests in the label queue. Merging-aware cache is organized as a traditional set-associative cache. Each cache line uses logical address (LA) as its tag, and data segment includes the decrypted data block, program address (PA), and label value (Label).

It is worth mentioning that components in the ORAM controller can function in parallel with the DRAM accesses. Due to the long access latency per ORAM request, most of the latencies induced by the controller are overlapped. In addition, some requests may complete before accessing DRAM because of the ORAM controller's caching effect.

5. EVALUATION

In this section, we first present experimental setup. The efficiency of Fork Path ORAM is then evaluated at a representative system configuration. Finally, sensitivity analysis of different configurations is provided.

5.1 Experimental Setup

Performance evaluation is conducted with a full system simulator gem5 [31] integrated with DRAMSim2. The detailed configuration of processor, ORAM controller, and main memory are summarized in Table 1. Energy consumptions of ORAM control logic and cache are generated from logic synthesis tool of Synopsys [32] and CACTI [33], respectively.

Table 1: Processor Configuration.

Core, on-chip cache	
Core type	out-of-order Alpha
Core number	4, 8-way issue
Core frequency	2GHz
L1 I/D cache	32KB/32KB, 2-way, LRU
L1 read/write	1/1-cycle
L2 cache	1MB shared, 8-way, LRU
L2 read/write	10/10-cycle
ORAM controller	
Controller clock frequency	2.0GHz
Data block size	64B
Data ORAM capacity	4GB (L = 24)
Block slots per bucket(Z)	4
Memory controller and DRAM	
Memory type	DDR3-1600
Memory channels	2
Peak bandwidth	12.8GB/s

DRAMSim2 [34] is used to model the detailed memory accesses of the ORAM tree. We derive the default latency and energy parameters of DDR3 from DRAMSim2. Similar to prior works [12, 18], two memory channels are adopted in the design. In order to maintain a low probability of stash overflow, a 50% memory utilization is presumed [18]. It means that a 8GB memory is needed to store 4GB data. In addition, to maximize the utilization of DRAM bandwidth, a sub-tree layout [18] is adopted.

Multi-programmed and multi-threaded workloads are selected from SPEC 2006 [35] and PARSEC [36] benchmark suites to ensure a comprehensive evaluation, respectively. For multi-programmed workloads, we carefully mix the benchmarks to represent various cases: We first partition all SPEC benchmarks into two groups: high ORAM overhead group (HG) and low ORAM overhead group (LG). Benchmarks in Mix1 and Mix2 are randomly selected from the LG, while benchmarks in Mix3 and Mix4 are from the HG. Benchmarks in Mix5 (Mix6) and Mix8 (Mix7) are randomly selected from LG (HG) to simulate the situation of duplicated programs. Benchmarks in Mix9 and Mix10 are randomly selected from both groups. We list the details of the multi-programs in Table 2.

Table 2: Mixed benchmarks from SPEC 2006

Mix1	453.povray, 458.sjeng, 459.GemsFDTD, 464.h264ref
Mix2	401.bzip2, 465.tonto, 471.omnetpp, 473.astar
Mix3	403.gcc, 410.bwaves, 429.mcf, 435.gromacs
Mix4	462.libquantum, 470.lbm, 481.wrf, 444.namd
Mix5	453.povray,453.povray, 458.sjeng, 458.sjeng
Mix6	444.namd, 444.namd, 435.gromacs, 435.gromacs
Mix7	410.bwaves, 410.bwaves, 410.bwaves, 410.bwaves
Mix8	464.h264ref, 464.h264ref, 464.h264ref, 464.h264ref
Mix9	454.calculix, 464.h264ref, 429.mcf, 458.sjeng
Mix10	401.bzip2, 453.povray, 462.libquantum, 462.libquantum

5.2 Evaluation with a Fixed Configuration

In this subsection, the efficiency of Fork ORAM Path is evaluated and compared to traditional ORAM design. To simplify discussion, we focus on the four-core configuration in Section 5.1 using multi-programmed workloads. Experiment results of other configurations will be presented in next subsection. We first evaluate effects of merging+scheduling and merging-aware caching with ORAM performance. A full system evaluation is then performed to retrieve the results of execution time and energy consumption.

5.2.1 ORAM Performance Evaluation

In Figure 10, the average length of ORAM tree path after applying path merging and request scheduling (labelled as 'merging') is compared to the baseline Path ORAM (labeled as 'Traditional ORAM') with different label queue sizes. Note that only path merging is applied when the label queue size is set to 1. The baseline path length always equals 25 because a complete path from the leaf to root must be traversed in traditional Path ORAM. It is easy to tell that the average length of the accessed ORAM path reduces when the label queue size increases. Statistically, the average ORAM path length is application-independent but decreases linearly with $log(Label\ Queue\ size)$.

Figure 10 also shows the reduction in DRAM latency of each ORAM request, which is directly impacted by the reduction in path length. In fact, the reduction of DRAM latency is even more significant than that of the path length. This is because the DRAM row-buffer miss rates also decreases with the length of ORAM path.

Here the results include both real data ORAM requests and dummy ORAM requests. As previously discussed, applying path merging and request scheduling

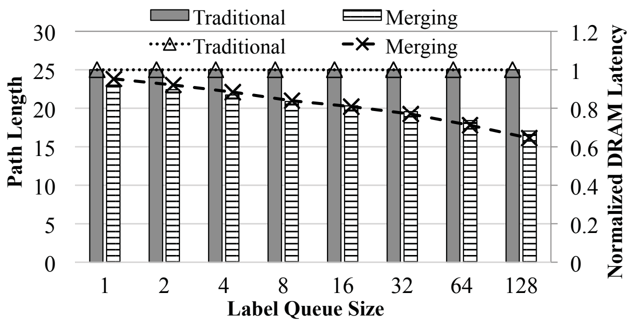

Figure 10: Average ORAM path length and average DRAM latency (marked as "Δ" and "×") with different Label Queue sizes.

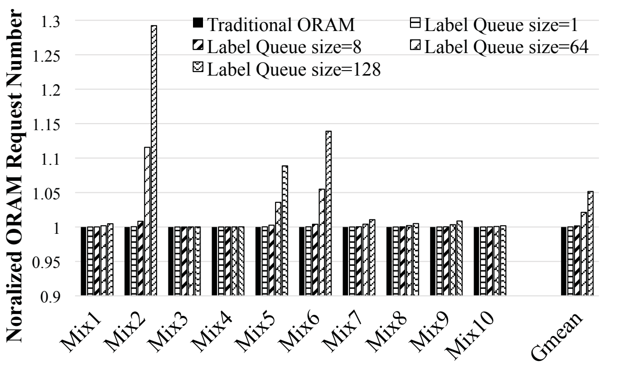

Figure 11: Normalized total number of ORAM requests.

can induces extra dummy ORAM requests. In Figure 11, total ORAM requests are normalized over baseline Path ORAM for comparison. We found that the number of extra ORAM requests increases with the Label Queue size. For most workloads, the increase is moderate because the dummy request replacing is applied. For some workloads, significant extra ORAM requests can be still observed (e.g., over 25% for Mix2). It is mainly because these workloads have really low memory intensity in some periods during execution. Thus, a lot of extra dummy ORAM requests are generated. On average, the total number ORAM requests is increased by only 5% even for a Label Queue size of 128.

Note that both path length and the number of dummy requests affect the ORAM performance. In order to provide a comprehensive and straightforward evaluation of ORAM performance, we introduce a metric called average data request ORAM latency (shorten as **ORAM latency**). It represents *the completion time of a LLC request since it enters the ORAM controller.* ORAM latency can reflect both the reduction in memory traffic and queueing latency, and directly reflect the overhead of ORAM.

The ORAM latencies of Fork Path ORAM with traditional ORAM are compared in Figure 12 with different label queue sizes. For most workloads, ORAM latency decreases at first as the queue size increases. However, when the queue size is increased from from 64 to 128, ORAM latency is increased. It means that the benefits

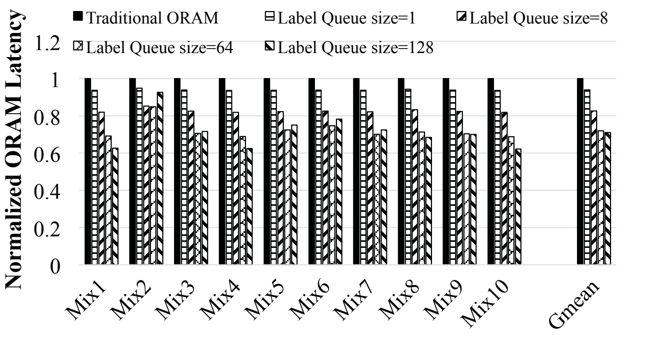

Figure 12: ORAM latency with different label queue sizes

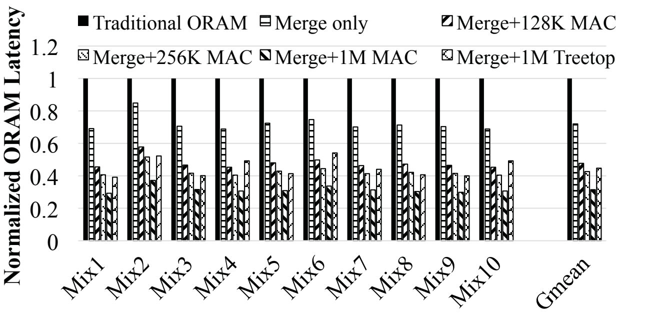

Figure 13: ORAM latency with different caching designs.

of path length reduction has been offset by the extra dummy requests induced by using such a large queue size. Thus, it is proper to set label queue size to **64**, which is used as default value in the rest of this work.

The efficiency of merging-aware caching is evaluated in Figure 13. Apparently, ORAM latency is reduced after using on-chip caching. Compared to prior treetop caching, merging-aware caching (labeled as MAC) can further reduce ORAM path length and consequently, achieve a further reduction in ORAM latency. We vary MAC sizes from $128K$ bytes to $1M$ bytes and compare them to the case using $1M$ bytes treetop caching. On average, using merging-aware caching can achieve a reduction in ORAM latency comparable to treetop caching with only about 1/4 of cache size.

5.2.2 *Full System Evaluation*

Figure 14 presents the results of the slowdown of program execution time, which is normalized to the insecure processor. The label queue size is set to be 64, as mentioned in Section 5.2.1. The cache size of MAC is set to $128KB$, $256KB$ and $1MB$ while the cache size of treetop caching is fixed at $1MB$. We can see that Fork Path ORAM improves the system performance significantly. When the cache size of MAC is $1MB$, system execution time reduces by 58% and 29%, compared to the traditional ORAM and that using a $1MB$ treetop caching, respectively.

Fork Path ORAM can also help reducing energy con-

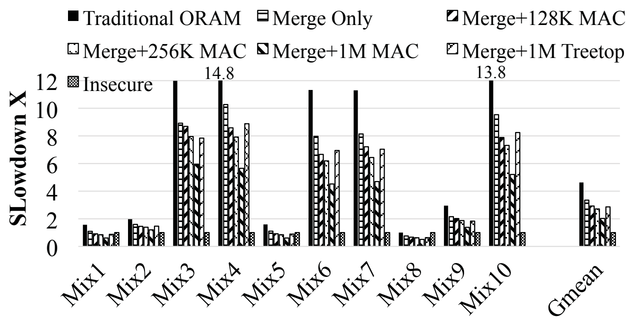

Figure 14: Slowdown of full system execution time.

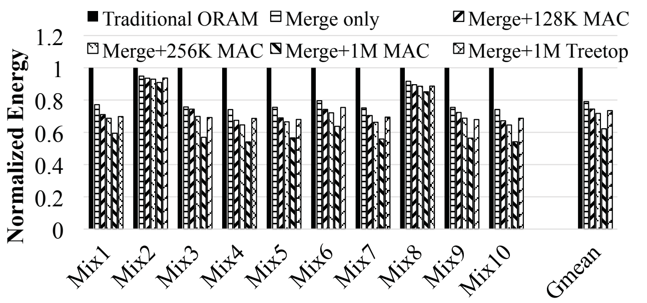

Figure 15: Energy consumption of ORAM memory system.

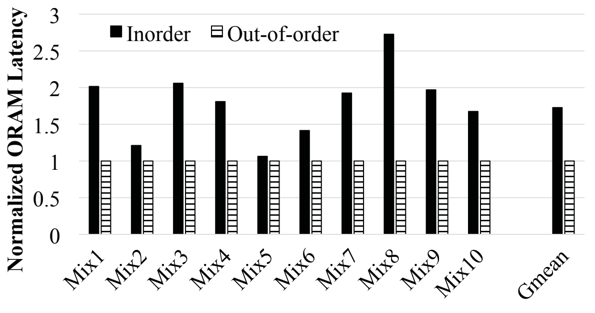

Figure 16: In-order vs. out-of-order

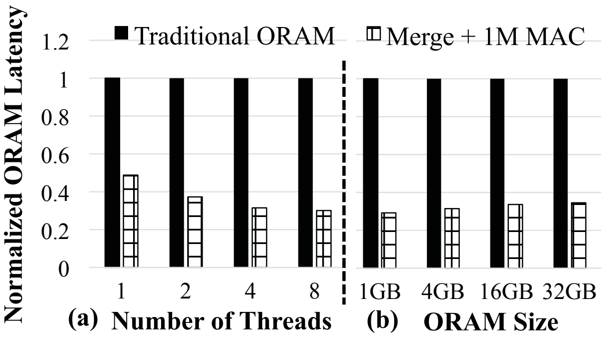

Figure 17: ORAM latency with (a) 1/2/4/8 thread(s) (b) different ORAM sizes

sumption of memory accesses. Total energy consumption of ORAM, including both external memory and ORAM controller, is shown in Figure 15. Although extra components are added in ORAM controller, the total energy consumption of ORAM is still reduced because it is dominated by the energy consumption of external memory system. On average, the energy consumption is reduced by about 38% compared to the traditional ORAM when both path merging/scheduling and 1*MByte* MAC are adopted. Even compared to the case using 1*MByte* treetop caching, we can still achieve 15% energy reduction.

5.3 Sensitivity Analysis

In this subsection, we evaluate efficiency of Fork Path ORAM with different system configurations and workloads for sensitivity analysis. Note that we fix the label queue size at 64 and set the capacity of merging aware cache to 1*MByte*. As previously discussed, we use ORAM latency to measure the ORAM efficiency.

Figure 16 compares the ORAM latency of in-order and out-of-order processors. The ORAM latency of in-order processor is significantly higher than that of out-of-order one. It is because of the low memory intensity in in-order execution. Hence, using in-order processor induces more extra dummy ORAM requests with the same label queue size of 64. The comparison further proves that Fork Path ORAM is more efficient when memory intensity is high. In fact, a smaller label queue size may be preferred for an in-order processor.

Figure 17(a) shows the ORAM latency when the number of thread varies. We simulate multi-program benchmarks, which are selected following the similar method in Table 2, on a processor with 1/2/4/8 cores. Then, the geometric mean of ORAM latency using Fork Path ORAM is normalized to that of traditional ORAM. We can find that, when the number of threads increases, the advantage of ORAM latency improves at the same time. It is mainly because the memory intensity increases with the thread number and hence, demonstrating an improved efficiency of ORAM.

Figure 17(b) presents the ORAM latency with different ORAM sizes. Here we fix the thread number to 4. Similarly, the geometric mean results are listed and compared to traditional ORAM with multi-programmed benchmarks in Table 2. We can find that, with the increase of ORAM size, the efficiency of ORAM latency is moderately degraded. It is because the length of a complete path from leaf to root of the ORAM tree increases with the ORAM size, but the reduction of ORAM path length is fixed with the same Fork Path ORAM design.

Figure 18 shows the ORAM latency at different numbers of DRAM channels. The results are all normalized to traditional ORAM with corresponding configurations. We can find that Fork Path ORAM is more efficient with less memory channels. It is because the absolute ORAM latency increases with less channels. Thus, the ratio of pending real data ORAM requests in the label queue increases accordingly.

Figure 19 shows the normalized ORAM latency of

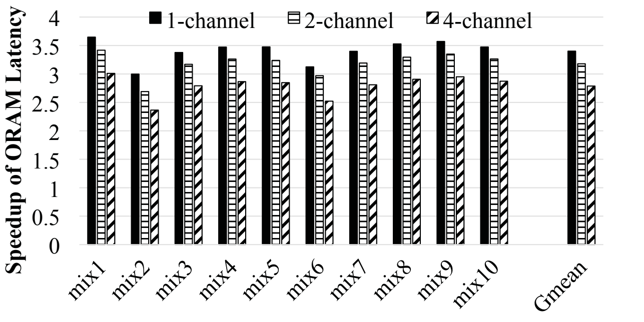

Figure 18: Speedup of ORAM latency with 1/2/4 DRAM channel(s).

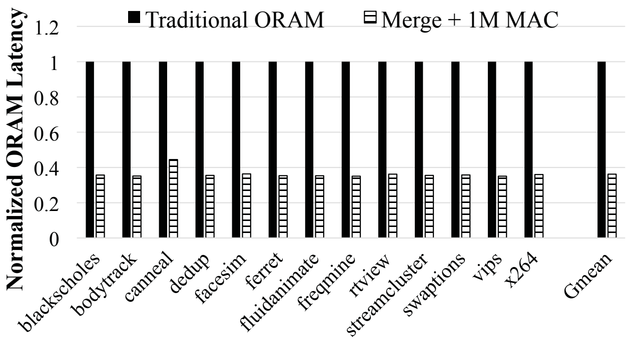

Figure 19: ORAM latency of multi-threading applications (4-thread).

multi-threading workloads from PARSEC. The number of threads is set to 4 so that one thread is running on each core. ORAM latency is also reduced significantly compared to baseline traditional Path ORAM. How much the ORAM latency can be reduced depends on the numbers of extra dummy ORAM requests that are related to memory intensity.

6. RELATED WORK

Oblivious RAM is first proposed by Goldreich and Ostrovsky [15, 14]. Since then, numerous follow-up works been been performed to improve its feasibility and efficiency [11, 10, 9, 16, 17]. Among these works, Path ORAM [10] has drawn wide attentions because of its simplicity and high efficiency and considered as one of the most promising protocols for a secure processor.

Ren *et al.* propose several optimization techniques for basic Path ORAM, such as background eviction, static super block, and subtree layout [18]. In background eviction, DRAM utilization, access overhead, and stash overflow possibility are all improved. In static super block, several adjacent program addresses are mapped to the same label. Hence, one path load may fulfill several requests because of locality. The subtree layout can reduce the row buffer miss when DRAM is applied to store an ORAM tree.

Maas *et al.* demonstrate Phantom [13] – the first hardware implementation of Path ORAM, in which tree-top caching and min-heap eviction are proposed to reduce the latency of path accesses and stash operations.

Fletcher *et al.* propose a dynamic scheme to protect the timing channel of ORAM accesses [25]. With limited leakage of information, their scheme reduces 30% performance degradation compared to a zero-leakage scheme with the same power consumption. Freecursive ORAM [12] is presented by the same group later where PosMap Lookaside Buffer (PLB) and PosMap compression are introduced to mitigate the overhead of PosMap accesses. Results show that 95% PosMap-related accesses to the memory are reduced.

Yu *et al.* propose PrORAM [19] in which dynamic prefetching is introduced. Compared to static prefetching (i.e., static super block), dynamic prefetching is more flexible to join or disjoin adjacent blocks according to the program's locality. Experiments demonstrate that PrORAM offers on average twice performance gain w.r.t. static prefetching.

7. CONCLUSION

ORAM has become an important component of modern secure processors, however, followed by significant overhead on memory accesses. By examining the state-of-art Path ORAM design, we find that there is a large volume of redundant ORAM memory requests that can be removed without harming the security. Based on this observation, we propose a novel Fork Path ORAM to remove these redundant accesses using the path merging and request scheduling techniques. Moreover, a merging-aware caching technique is developed to improve caching efficiency. Our results show that applying Fork Path ORAM can substantially reduce the overhead of memory accesses, resulting in significant system performance enhancement especially in the case where memory intensity is high.

8. ACKNOWLEDGEMENTS

We thank anonymous reviewers for their valuable comments and feedback. This work was supported by NSF China 61572045.

9. REFERENCES

[1] 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, no. 11, pp. 168–177, 2000.

[2] 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*, pp. 160–171, ACM, 2003.

[3] 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*, pp. 33–44, ACM, 2015.

[4] 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*, pp. 14–24, IEEE Computer Society, 2005.

[5] 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*, pp. 295–306, IEEE, 2003.

[6] 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, pp. 72–84, ACM, 2004.

[7] 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.

[8] 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.

[9] 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*, pp. 157–167, SIAM, 2012.

[10] 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*, pp. 299–310, ACM, 2013.

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

[12] 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*, pp. 103–116, ACM, 2015.

[13] 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*, pp. 311–324, ACM, 2013.

[14] 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*, pp. 182–194, ACM, 1987.

[15] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[16] 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*, pp. 197–214, Springer, 2011.

[17] 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*, pp. 143–156, SIAM, 2012.

[18] 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, pp. 571–582, ACM, 2013.

[19] 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*, pp. 616–628, ACM, 2015.

[20] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 351, IEEE Computer Society, 2003.

[21] W. Shi and H.-H. S. Lee, "Authentication control point and its implications for secure processor design," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 103–112, IEEE Computer Society, 2006.

[22] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 339, IEEE Computer Society, 2003.

[23] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pp. 221–232, IEEE, 2004.

[24] 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*, pp. 183–196, IEEE Computer Society, 2007.

[25] 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)*, pp. 213–224, IEEE, 2014.

[26] C. Yan *et al.*, "Improving cost, performance, and security of memory encryption and authentication," in *33rd International Symposium on Computer Architecture (ISCA'06)*, pp. 179–190, 2006.

[27] R. Callan, A. Zajic, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 242–254, IEEE, 2014.

[28] 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*, pp. 216–228, IEEE, 2014.

[29] F. Liu and R. B. Lee, "Random fill cache architecture," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 203–215, IEEE, 2014.

[30] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pp. 3–8, ACM, 2012.

[31] 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, pp. 1–7, Aug. 2011.

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

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

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

[35] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[36] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81, ACM, 2008.