International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 7, Number 2, pages 106–123, July 2017

A Virtual Cache for Overlapped Memory Accesses of Path ORAM

Naoki Fujieda, Ryo Yamauchi, Hiroki Fujita and Shuichi Ichikawa
Department of Electrical and Electronic Information Engineering,
Toyohashi University of Technology, Toyohashi, Aichi, 441-8580, Japan

Received: January 31, 2017
Revised: April 19, 2017
Accepted: May 26, 2017
Communicated by Michihiro Koibuchi

Abstract

Oblivious RAM (ORAM) is a technique to hide the access pattern of data to untrusted memory along with their contents. Path ORAM is a recent lightweight ORAM protocol, whose derived access pattern involves some redundancy that can be removed without the loss of security. This paper presents *last path caching*, which removes the redundancy of Path ORAM with a simpler protocol than an existing method called Fork Path ORAM. By combining *Delay* and *Reuse* schemes, the performance of our technique was comparable with Fork Path ORAM. According to our evaluation with a prototyped FPGA implementation, the number of LUTs used with the last path caching was 1.4%–7.8% smaller than Fork Path ORAM.

1 Introduction

Memory encryption [7] is a common technique for secure processors to prevent information leakage from a data bus to an external memory being observed [15, 17]. However, information does not only be leaked from the data themselves, but also from their access pattern, i.e. the sequence of memory addresses accessed by the processor [8]. For example, some of search queries to an email repository [8] and an SQLite database [10] can be distinguished by observing access pattern. Memory encryption hides the contents, while it cannot hide the access pattern.

Oblivious RAM (ORAM) [6] is a technique to hide the data and their access pattern by shuffling data and adding dummy memory accesses. Path ORAM [14] is a recent lightweight ORAM protocol, which was proposed with a hardware implementation called PHANTOM [10]. However, its overhead on the bandwidth is still high for practical use.

This study focuses on the redundancy of Path ORAM. A **path** is a set of memory blocks read and written through an ORAM access. In Path ORAM, two consecutive paths has an overlapped region. Writing to and reading from such regions can be removed as redundant memory accesses without the loss of security.

To deal with this redundancy, this paper introduces *last path caching*, which has a simpler procedure than an existing scheme, called Fork Path ORAM [20]. This paper also points out that Fork Path ORAM has an disadvantage on security: the derived access pattern may reflect the original access pattern in a specific condition. Though it can be solved in some ways, the efficiency and simplicity will decrease. Last path caching, on the other hand, is a virtual cache mechanism using an existing cache in Path ORAM called a stash and it can be implemented with a minimal modification to the Path ORAM protocol.

We have presented a preliminary version of this work in CANDAR 2016 [4]. The differences from the preliminary work include:

- possible solutions to a security-related problem of Fork Path ORAM and an evaluation of them (in Section 3.3 and 3.4),
- a hardware implementation of the last path caching and its evaluation with a prototype of ORAM controller (in Section 6),
- a detailed evaluation of the combination with another existing virtual cache mechanism, called treetop caching (in Section 7), and
- an explanation and a discussion about Merging Aware Cache, which was proposed with Fork Path ORAM [20] (in Section 7).

We also rename the schemes of the last path caching from WT and WB to Reuse and Delay, respectively, after the paper we first introduced the concept [5].

The rest of this paper is organized as follows: Section 2 provides an overview and a working example of Path ORAM. Section 3 includes an explanation of redundancy of Path ORAM and an existing technique used in Fork Path ORAM, along with its shortcomings, possible solutions to them, and evaluation of Fork Path ORAM. Section 4 presents our last path caching and its several schemes. We evaluate the performance of ORAM systems in Section 5. A hardware implementation of the last path caching and its evaluation is described in Section 6. We discuss some related studies about optimization techniques and ORAM architectures in Section 7. Finally, we conclude the paper in Section 8.

2 Path ORAM

2.1 Organization of Path ORAM

Path ORAM [20] is a lightweight ORAM protocol, and PHANTOM [10] is the first hardware implementation of Path ORAM on FPGAs. The main data structure of Path ORAM consists of an ORAM tree, a stash, and a position map, shown in Figure 1 (a).

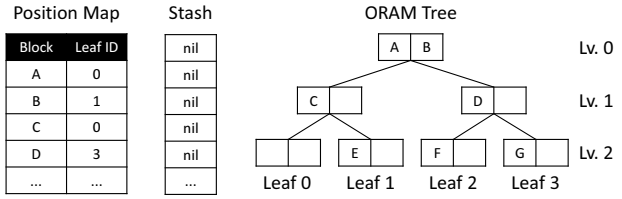
The **ORAM tree** is a binary tree of encrypted data, which is mapped to an external, untrusted memory. Assume the number of blocks storing actual data to be N, the height of the tree L is set to approximately $\log_2 N$, and the levels are numbered from level 0 (root) to level L (leaf). Each leaf has its own ID from 0 (left) to $2^L - 1$ (right). Each node, which is sometimes called a bucket, holds Z blocks. The number of blocks in the ORAM tree, including dummy data, is calculated as $(2^{L+1} - 1)Z$. The ORAM tree is accessed based on a path from the root to a leaf. In this paper, the path to the leaf x is denoted as $\mathcal{P}(x)$.

The **stash** is an on-chip cache of the ORAM tree that temporarily keeps blocks being read from the tree. Some blocks might not be written back to the tree during an ORAM access. The stash must be large enough that the probability of shortage of the stash due to such blocks is negligible. In order to determine which block should be written back, the stash has to keep the leaf IDs of its entries.

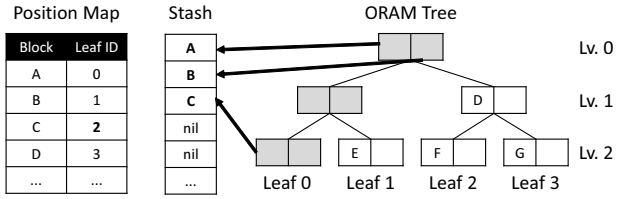
The **position map** is a lookup table between the block address given from the processor and the path (or leaf ID) that the corresponding block belongs. A block with a leaf ID of x is found either in the buckets on $\mathcal{P}(x)$ or in the stash. The capacity of the position map is NL bits. If it is too large to be stored on the chip, a recursive approach is applied where the position map is managed by another Path ORAM [20].

2.2 ORAM Access

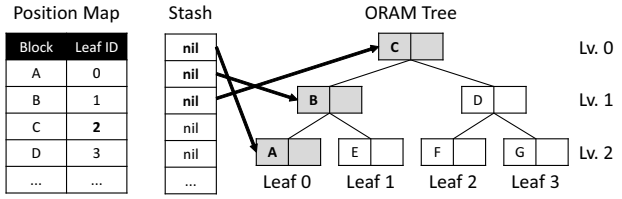
An ORAM access in Path ORAM is divided into four steps. It is also illustrated with an example in Figure 1, where the requested block from the processor is B. This example assumes that $L = 2$ and $Z = 2$. In the ORAM tree, actual data blocks are labeled by A, B, ..., and G, while dummy



(a) Initial status.



(b) Step 3: reading $\mathcal{P}(0)$ for block C.



(c) Step 4: writing $\mathcal{P}(0)$ back.

Figure 1: Organization and working example of Path ORAM.

blocks are denoted by blanks. Before a request comes, blocks A and B are placed at the root and C is in the left node of level 1. Blocks A, B, C, and D have their respective leaf IDs of 0, 1, 0, and 3.

Step 1 is a search for the stash. If the requested block is found in the stash, it is immediately accessed by the processor and the following steps are omitted. In this example, this step has no effect because there are no valid blocks in the stash.

Step 2 is a lookup and an update of the position map. The leaf ID of the requested block (x in the following steps) is read from the corresponding entry in the position map. It is then replaced by a random number from 0 to $2^L - 1$ so that the block can be remapped to another path. In the example, this step reveals that the block C is stored on $\mathcal{P}(0)$. It will be remapped to $\mathcal{P}(2)$ after the ORAM access.

Step 3 is a read of a path. All blocks in the nodes on $\mathcal{P}(x)$ are read from external memory and decrypted. Actual data blocks are moved to the stash. Now that the requested block is in the stash, it is accessed by the processor. In Figure 1 (b), all blocks on $\mathcal{P}(0)$, painted in gray, are read and the blocks A, B, and C are stored into the stash.

Step 4 is a write back of the path. It applies the following processes to all the nodes on $\mathcal{P}(x)$, in order from the leaf to the root. First, it picks out blocks in the stash whose path includes the target node. If Z or more blocks are extracted, Z blocks chosen from them are encrypted and written to external memory. Otherwise, all the extracted blocks and dummy block(s) are written after encryption. To put it briefly, blocks in the stash are written back to buckets as near as possible to the leaf. These processes can be done by a min-heap that reorders stash entries with a bit-wise XOR of each leaf ID and the current path's ID [10]. The block with the smallest XOR value has the largest overlap with the current path. In Figure 1 (c), blocks in the stash are reordered with the XOR values in advance. They are sorted in the order of A, B, and C. At first, blocks with paths that includes leaf 0 are extracted. Since only $\mathcal{P}(0)$ includes leaf 0, only the block A is picked. The block A and a dummy block are written back to leaf 0. The left node of level 1 is included by $\mathcal{P}(0)$ and $\mathcal{P}(1)$. The block B and a dummy block are written there. The remaining block C is written

to the root with a dummy block. Note that some blocks might remain in the stash after this step, though all blocks are written back to the ORAM tree in this example.

With these steps, a request from the processor (that misses in the stash) becomes a sequence of a read from and a write to a random path. Therefore, the access pattern derived from Path ORAM is a sequence of accesses to random paths, which is completely independent of the original access pattern from the processor.

3 Redundancy of Path ORAM

3.1 Example of Redundant Access

Suppose a request to the block E comes after the example shown in Figure 1 (c). Since the block E is located in the leaf 1, all blocks along $\mathcal{P}(1)$ is read from external memory. The point is that the blocks read from the root and the left node of level 1 were written back and removed from the stash in the previous ORAM access. If they remained valid on the chip, the access to these nodes would be omitted. If the block B, instead of the block E, were requested under the same condition, the whole access to the ORAM tree would be omitted because the block B was read to the stash in the previous ORAM access.

An observation from this example is twofold: (1) the paths of two successive ORAM accesses overlap to some extent, and (2) the preceding path-write-back and the subsequent path-read of the overlapped part are redundant. The sequence of the accessed paths is "public information that visible to anyone" [10] and it has no information about the original access pattern. The redundant parts of ORAM access can be safely removed as far as the way of removal is also independent of the original access pattern.

3.2 Path Merging

Fork Path ORAM [20] was proposed by Zhang et al. to remove the redundant memory accesses in Path ORAM. The basis of Fork Path ORAM is that, if the path that will be accessed in the next ORAM access is known before the path-write-back step, the overlapped part between the current and the next paths can be determined.

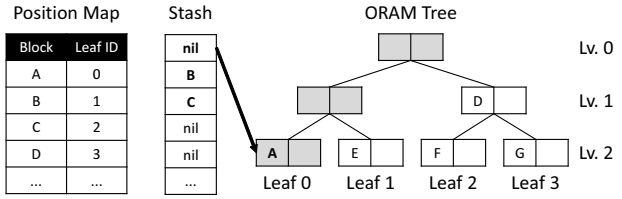
Figure 2 shows an example of Fork Path ORAM, where a path-read from $\mathcal{P}(0)$ has been completed and the next ORAM request to the block E has arrived. Since the block E belongs to $\mathcal{P}(1)$, the non-overlapped parts between the two paths are the leaf 0 and the leaf 1. In the path-write-back, only leaf 0 is written back (Figure 2 (a)). Similarly, only leaf 1 is read from the ORAM tree in the next path-read (Figure 2 (b)).

If the next requested block is not known, a dummy request is inserted. In the original Path ORAM, it only has to be known before completing the path-write-back. This difference causes extra dummy requests in Fork Path ORAM, which may affect the performance. However, even though a dummy request has been inserted, there is a chance to replace it silently with an incoming real request. At the time a real request comes, it can be safely replaced if the series of write-back access that has been issued is the same as the sequence that would have been issued when it had been inserted from the beginning. This supplementary technique is called dummy label replacing [20].

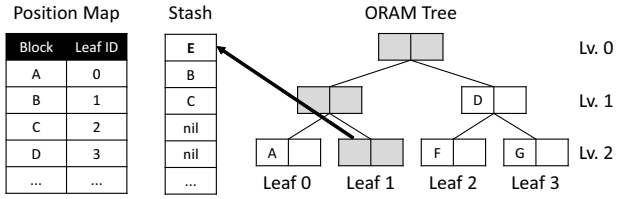
To increase the overlap of paths, Fork Path ORAM also adapts a reordering of ORAM requests. It has a request window with a fixed number of ORAM requests. The request that has the nearest path to the currently accessed path in the window is selected as the next request. If the number of real requests is smaller than the window size, dummy requests are inserted to fill the window.

3.3 Weaknesses of Fork Path ORAM and Possible Solutions

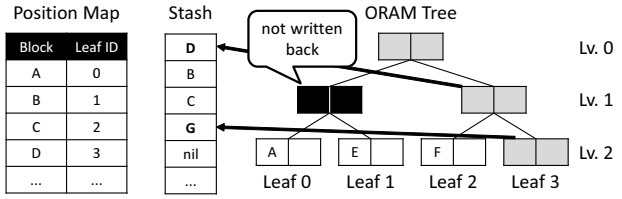
Although Fork Path ORAM completely removes the access to the overlapped parts of paths, it has two major weaknesses. The first one arises from property of the dummy requests. An application with a small number of memory request is more sensitive to the access latency and easily affected by the dummy requests. Its performance will be much worse by adapting Fork Path ORAM due to



(a) Writing non-overlapped part of $\mathcal{P}(0)$.



(b) Reading non-overlapped part of $\mathcal{P}(1)$.



(c) Problem after a hit of the block B in the stash.

Figure 2: Working examples of path merging in Fork Path ORAM.

extra dummy requests. The dummy label replacing technique solves the problem to some extent; however, it must be carefully considered that it is worth the cost of making the ORAM controller much more complicated.

The second weakness, which is more critical, security-related problem, is the loss of the determinacy of the derived access pattern. Though it can be solved in some ways as we describe later in this section, dealing with it harms the performance and the simplicity of the Fork Path ORAM. This leaves room for us to consider a simple alternative. We evaluate the effect of possible solutions on the performance in Section 3.4. Suppose the blocks B and D are requested in order after Figure 1 (b). The write back step is the same as the previous example shown in Figure 2 (a). Since the block B is found in the stash, the controller then skips the path access for the block B and starts to search for the block D. The only overlapped node between the previous path ($\mathcal{P}(0)$) and the block D's path ($\mathcal{P}(3)$) is the root. The non-overlapped part, the left node of level 1 and the leaf 3, are read to the stash (Figure 2 (c)). The problem is that the left node of level 1, shown in black, has not been written back. It comes from skipping the path access for the block B. That node is included in the block B's path ($\mathcal{P}(1)$) but not in the block D's path ($\mathcal{P}(3)$). If only the block D had been requested after Fig 1 (b), the black node would have been written back! It leaks the information that there was a hit in the stash on the way, which may be a hint for the original access pattern from the processor.

There are three possible solutions to the latter weakness. The first solution is to ignore hits in the stash. Even though the requested block is found in the stash, the path access for that block is continued there in order to avoid the problem by skipping it. It obviously degrades the performance due to unnecessary path accesses. We call this solution *None* for the stash is not checked for a hit at all. In the case of Figure 2, the path access for the block B is not skipped as if it was not found in the stash. As a result, the non-overlapped part with $\mathcal{P}(0)$ (i.e. the leaf 1) is read and the non-overlapped part with $\mathcal{P}(3)$, including the black node, is written.

The second solution is to exclude all requests that hit in the stash from the path merging. The

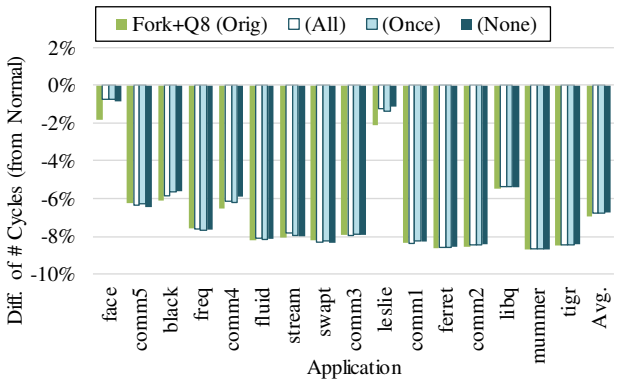


Figure 3: Result of preliminary evaluation of Fork Path ORAM. The size of reordering window is set to 8 (Q8).

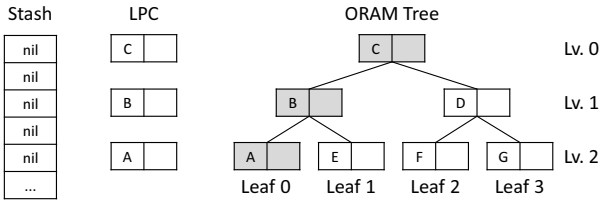
problem can also be avoided if a stash hit never occurs in the first step of ORAM access (searching for the stash). Though it may be a natural assumption, as far as we understand, Fork Path ORAM does not consider it because this step is left unchanged [20]. The stash is checked for a hit in two cases. The first case is when the path-read step has been completed and subsequent requests had already arrived. In this case, the stash may be checked repeatedly until a request that misses in the stash is found or the request queue becomes empty. The second case is when a new request, which might replace a dummy request using the dummy label replacing, comes in the path-write-back step. We call this solution *All* for all of the incoming requests cause searches for the stash. In Figure 2, the request for the block B will be processed immediately as the block is found in the stash. Now that the next request is for the block D, the non-overlapped part of $\mathcal{P}(0)$ with $\mathcal{P}(3)$, which includes the black node, is written back in the path-write-back step for $\mathcal{P}(0)$. It makes an ORAM controller much complicated because a search for the stash is required every time a new request comes to the head of a request window.

The third solution is a combination of the other two solutions. It searches for the stash and excludes a stash-hitting request only once on the completion of the path-read step (i.e. the first case of the solution *All*). Note that only the first request is checked for exclusion, even though more than one requests have arrived. In all other cases, the stash is not checked and thus hits in the stash are ignored. It may balance the performance and the complexity of controller. In the case of Figure 2, it behaves differently according to when the request for the block B comes. If it comes before the completion of the previous path-read step, it will be excluded and the behavior will be the same as the solution *All*. Otherwise, the stash hit of the block B will be ignored and the behavior will be the same as the solution *None*.

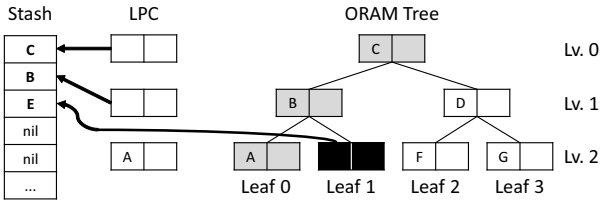
3.4 Evaluation of Fork Path ORAM

In this section, we make a preliminary evaluation about the possible solutions to a security-related weakness of Fork Path ORAM. The simulation environment and the performance index for this evaluation are the same as what we will explain in Section 5.1. The following four variants of Fork Path ORAM are evaluated here:

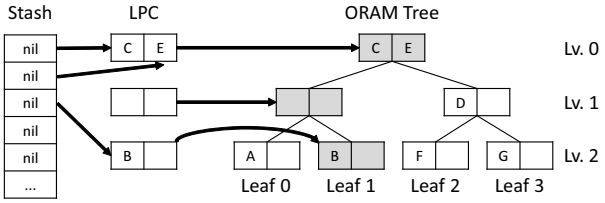
- the original Fork Path ORAM (*Orig*) that has a security-related weakness,
- the second solution that excludes stash hits from the path merging (*All*)
- the third solution that checks the stash only once (*Once*), and
- the first solution that ignores hits in the stash (*None*).



(a) Last Path Cache between the ORAM tree and the stash.



(b) Reading $\mathcal{P}(1)$ for the block E.



(c) Writing $\mathcal{P}(1)$ back, along with the LPC.

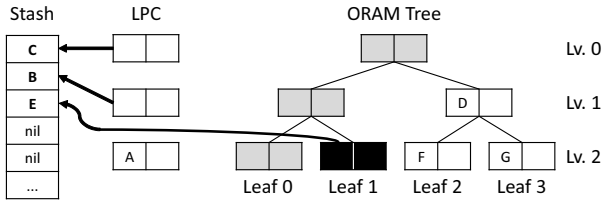
Figure 4: Organization and behavior of last path caching (*write-through*).

The possible solutions are named after the number of stash checks.

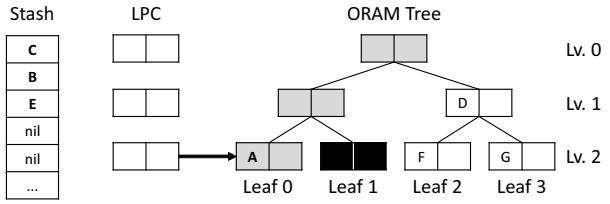
Figure 3 shows the relative execution time to the original Path ORAM [14]. The X-axis is the shortened name of application trace, while the Y-axis is the difference of the relative number of executed cycles. Avg. means the average of all of the traces. The performance loss with the possible solutions was about 0.2 percentage points on average. Even though all of the stash-hitting requests are excluded from the path merging, it sometimes falls into a situation that no real requests follow after an excluded request, which results in an increase of dummy requests. While the performance of *None* is slightly lower than the others, *All* and *Once* showed almost the same performance. We will apply *Once* to Fork Path ORAM in the following evaluations, considering the complexity of ORAM controller.

4 Last Path Caching

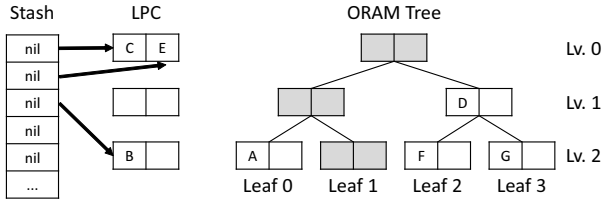
In this paper, we propose last path caching as an alternative technique to remove the redundant memory accesses, which has a simpler procedure than Fork Path ORAM. Figure 4 (a) illustrates its principles of operations. An additional cache called Last Path Cache (LPC) is placed between the ORAM tree and the stash. Though it can be implemented as a virtual cache in the stash as described in Section 6, we place an actual cache in the figures to simplify an explanation. It stores the previously written path from the stash. It has a function to mask or postpone some parts of path access. Its derived access pattern depends only on the sequence of accessed paths and it is independent of the original access pattern. Therefore the LPC does not harm the security of the Path ORAM. Although it can be write-through or write-back, it is shown as a write-through cache in Figure 4 (a). The characteristics of the last path caching vary according to the type of cache. In the following subsections, we introduce three schemes: Reuse, Delay, and Delay/Reuse Hybrid.



(a) Reading $\mathcal{P}(1)$ for the block E.



(b) Writing unread LPC blocks to the ORAM tree.



(c) Writing stash blocks to the LPC.

Figure 5: Behavior of the last path caching (*write-back*).

4.1 Reuse Scheme

Figure 4 shows the organization and a working example of the Reuse scheme. There, all levels of the LPC is set to write-through. It means that all the blocks in the LPC have been also written to the ORAM tree and they are present in the tree. In the example, if the block E is requested (Figure 4 (b)), the overlapped part (including B and C) is supplied from the LPC, while the non-overlapped part (including E) is read from the ORAM tree. The non-overlapped part in the LPC (including A) can be simply discarded. On the write back, the whole path are written to both the ORAM tree and the LPC, as shown in Figure 4 (c).

An advantage of the Reuse scheme is that they do not change the order of requests to external memory. As we will explain in Section 4.2, reordering the requests to external memory may affect the locality of access to external memory.

A weakness of the Reuse scheme is that the redundant write accesses are not removed. It means the reduction of the memory accesses becomes only a half of that achieved by Fork Path ORAM, when ignoring the effect of the extra dummy requests. The effect of this weakness gets large in nodes near the root. Such nodes are more likely to be included in the overlapped region. On the other hand, nodes near the leaf are little affected. They are not likely to be overlapped and they will have to be written back to the ORAM tree after all.

4.2 Delay Scheme

Figure 5 describes the Delay scheme, where all levels of the LPC is set to write-back. All blocks in the LPC are dirty: they do not exist in the ORAM tree and they must be written either to the tree or to the stash before the path-write-back step of the subsequent access. The read step (Figure 5 (a)) works in the same way as the Reuse scheme. The overlapped part in the LPC are read to the stash, while the non-overlapped part must be written to the ORAM tree. Before the write-back

Table 1: Traces for evaluation, which are sorted by MPKI (Miss Per Kilo Instruction) of the last level cache.

Name	MPKI	Name	MPKI
face (facesim)	1.22	comm3	6.41
comm5	1.51	leslie (leslie3d)	7.75
black (blackscholes)	1.79	comm1	9.09
freq (freqmine)	2.69	ferret	13.00
comm4	3.74	comm2	14.99
fluid (fluidanimate)	4.09	libq (libquantum)	19.16
stream (streamcluster)	4.90	mummer	24.23
swapt (swaptions)	5.80	tigr	32.40

step, all blocks (including dummy blocks) in the non-overlapped part of the LPC are written back to the ORAM tree (Figure 5 (b)). Blocks in the stash are then written back to the LPC, rather than the ORAM tree (Figure 5 (c)). From the viewpoint of the derived access pattern with the Reuse scheme, it corresponds to a postponement of the path-write-back until the next ORAM access.

An advantage of the Delay scheme is that it does not suffer from extra dummy requests. The write-back step of Fork Path ORAM [20] requires the overlapped region between the current and the next path, while the Delay scheme needs the overlapped region between the current and the previous path. As well as the original Path ORAM, the next path has to be known before the completion of the path-write-back step in the Delay scheme.

A weakness of the Delay scheme is that the time between reading a block in the ORAM tree and writing back to the same block gets longer. DRAM is usually used as external memory, which leverages locality of access with a row buffer. Writing blocks immediately after reading increases the chance for the blocks to hit in the row buffer. The delay of the write back decreases the possibility of the row buffer hit, which degrades the effective bandwidth of the memory. This weakness gets more serious in nodes near the leaf. In the original Path ORAM, the path-read step begins from the root and the path-write-back step begins from the leaf. Thus the nodes near the leaf are written back with a shorter interval and more likely to hit in the row buffer. Conversely, the nodes near the root are not likely to hit in the row buffer and the effective memory bandwidth is not much affected even if their write-backs are delayed.

4.3 Delay/Reuse Hybrid Scheme

The summary of the weak points of the above-mentioned schemes is that the Reuse scheme does not affect much near the leaf and the Delay scheme has little effect near the root. The last scheme, called Delay/Reuse hybrid scheme, combines them to complement each other. To be more precise, with a threshold t, a part of the LPC from the root to the level $t-1$ is set to write-back and the rest (i.e. from the level t to the leaf) is set to write-through. The path-read step is in common with the Delay and Reuse schemes. The path-write-back step is different in the selection of blocks to be written to the ORAM tree. We will explain its detail along with a hardware implementation in Section 6.

5 Evaluation

5.1 Methodology

In this section, the performance of the proposed schemes is evaluated with a trace-based simulation environment. Sixteen traces from single-thread applications of the Memory Scheduling Championship [16] are used, Applications, listed in Table 1, are composed of PARSEC, SPEC CINT2006, BioBench, and commercial (*comm*) workloads. Requests in the traces have been filtered in advance by a last level cache with 512 kiB of capacity and 64-byte data blocks [1]. They are simulated with a modified version of a DRAM simulator usimm [1] where an ORAM controller is added between

Table 2: System Parameters.

Processor Cores	
Core Type	4-way, out-of-order
# of Cores	4
Core Frequency	3.2 GHz
Last-level Cache	512 kB/core
DRAM and Memory Controller	
# of DRAM Channels	4
DRAM Frequency	800 MHz
Peak Throughput	51.2 GB/s
ORAM Controller	
Data Block Size	64 B
Height of Tree (L)	23
# of Slots per bucket (Z)	4
# of Levels for Treetop Cache	3
ORAM Hit Latency	40 ns
AES Circuit Latency	25 ns
AES Circuit Throughput	204.8 GB/s

Table 3: Result of evaluation without reordering.

Criterion	Delay	Reuse	D/R	Fork
# of Cycles	+0.6%	-2.2%	-3.9%	-4.0%
# of DRAM Accesses	-8.3%	-4.2%	-8.3%	-8.3%
DRAM Row Buffer Miss Rate on Write	+89.9%	0.0%	+7.3%	+7.3%

the processor and the DRAM controller. Main system parameters are summarized in Table 2. All four cores executes the same program. The first 400 million instructions are fast-forwarded to warm ORAM up. The following 100 million instructions are used for the measurement. The performance index is the sum of the number of cycles to execute them. The number of ORAM accesses, the number of DRAM accesses, and the miss rate in row buffers of DRAM on write are also measured for reference.

The following five ORAM variants are evaluated:

- the original Path ORAM [14] (*Normal*) as the baseline of performance,
- the Delay scheme (*Delay*),
- the Reuse scheme (*Reuse*),
- the Delay/Reuse hybrid scheme (*D/R*), and
- Fork Path ORAM [20] (*Fork*) for the comparison, where the *Once* solution in Section 3.4 is applied.

In the Delay/Reuse hybrid, the threshold is set to 8 unless otherwise mentioned: the LPC blocks from the root to the level 7 are set to write-back. We also evaluate a case of adapting a reordering of ORAM requests [20]. The size of the window is set to 4 or 8 requests, which is denoted as Q4 or Q8, respectively. In order to prevent starvation, the movement to find the next path is limited to one-way: the next request is selected from those with higher leaf ID than the current one. If no such requests are found, a request with the smallest leaf ID is selected.

5.2 Evaluation Result without Reordering

Table 3 summarizes the evaluation results in the case the reordering of requests is not applied. The results are shown by the relative difference from *Normal*. The average values of all the traces are

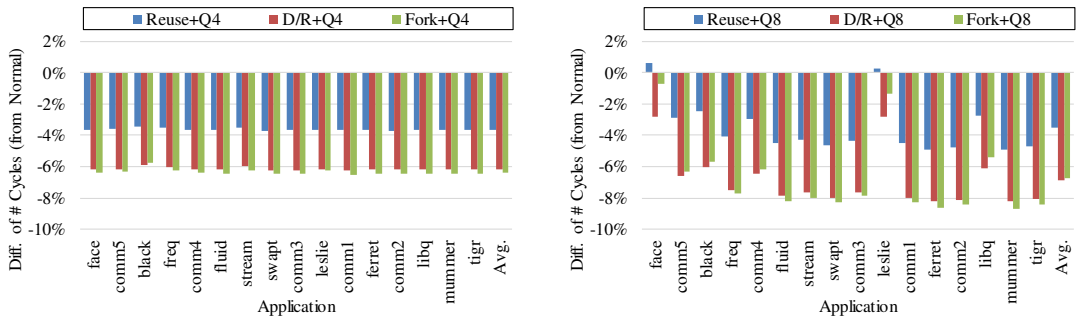


Figure 6: Result of performance evaluation with reordering window size of 4 (Q4) and 8 (Q8).

shown because they show almost the same result. The Delay scheme decreased the performance by 0.5%. While the number of DRAM accesses was reduced, the miss rate in row buffers was almost doubled, which widely increased the time to process an ORAM request. In the Reuse scheme, the performance gain was 2.2%, almost half of that in the Delay/Reuse hybrid or Fork Path. The reduction in DRAM accesses was also halved. This result confirms the discussion about the weakness of the Reuse scheme in Section IV. The result of the Delay/Reuse hybrid almost matched that of Fork Path. The increased row buffer miss came from the omission of access to levels near the root. Such blocks are also likely to hit in row buffers because they fit in a single row buffer per DRAM channel.

5.3 Evaluation Result with Reordering

Figure 6 shows the relative execution time where the reordering of requests is applied. Graphs (a) and (b) correspond the cases where the size of the window is 4 and 8, respectively. The X-axis is the shortened name of trace, while the Y-axis is the difference of the relative number of executed cycles. Avg. stands for the average of all the traces. The result of the Delay scheme is omitted in this evaluation.

When the window size was 4 (Figure 6 (a)), Almost the same result among the applications was observed, in similar to Table 3. The average performance loss of *D/R+Q4* over *Fork+Q4* was only 0.2 percentage points.

On the other hand, when the window size increased to 8 (Figure 6 (b)), the difference of the relative performance among the applications was shown. The performance gain got smaller in the applications that had fewer cache misses, with exceptions of *leslie* and *libq*. The advantage of the Delay/Reuse hybrid scheme over Fork Path was observed in such applications. With respect to the average increase of performance, *D/R+Q8* (6.87%) was slightly better than *Fork+Q8* (6.75%).

Detailed analysis on the difference of the number of DRAM accesses was made in Figure 7. Dark bars correspond to the difference (increase) of the number of ORAM accesses, while light bars mean the difference (decrease) of the number of DRAM accesses per ORAM access. The product of two numbers makes the number of DRAM accesses.

The increase of ORAM accesses came from dummy accesses due to the shortage of requests from the processor. The number of dummy requests became large when there were few cache misses (i.e. an application on the left, *leslie*, or *libq* was running). Dummy requests with Fork Path increased by 0.6 points on average (from 1.5% to 2.1% of real ORAM requests) over the proposed schemes. The increased dummy requests came from earlier deadline to decide the next path, which had been pointed out in Section 3.2. In other words, the situation that a dummy request cannot be replaced with a real request occurred at a frequency of 0.6% on average.

The number of DRAM accesses per ORAM access corresponded to the number of the redundant memory accesses actually omitted. Its difference between *D/R* and *Fork* was only 0.1 points on average. When the paths of two successive requests went too close, their overlapped region was

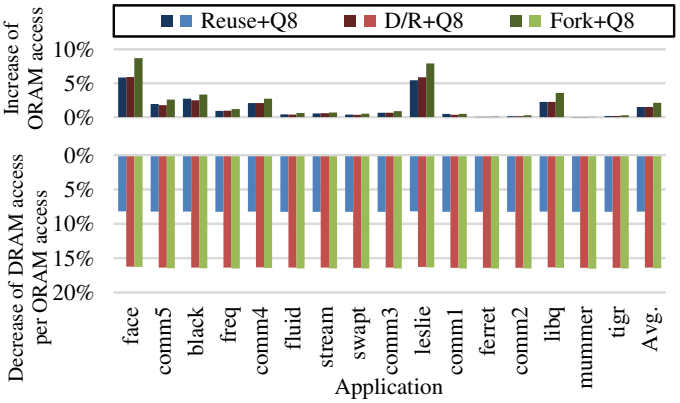
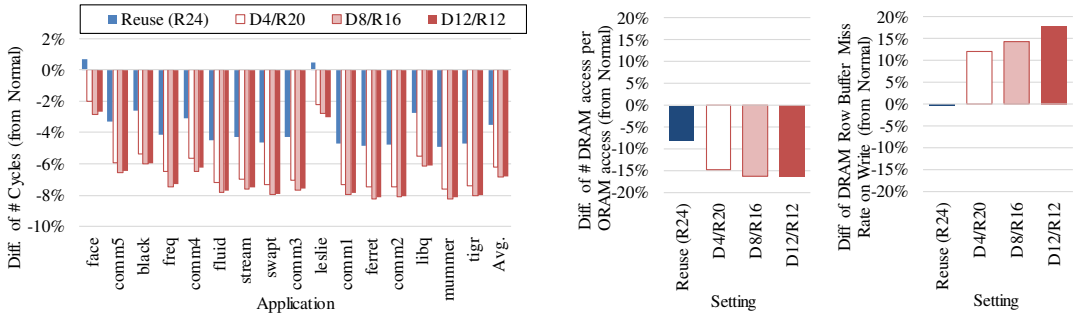


Figure 7: Breakdown of the difference of the number of DRAM access.



(a) The difference of the number of cycles.

(b) The difference of indices in DRAM access.

Figure 8: The effect of the threshold in Delay/Reuse Hybrid Scheme.

partially stored in the write-through levels of the LPC where write to external memory could not be removed. However, the result implied that it was rare case.

5.4 Effect of Threshold in Hybrid Scheme

Figure 8 depicts the result of a sensitivity study to the threshold of the Delay/Reuse hybrid scheme. Similarly to Figure 6, Graph (a) shows the difference of the number of executed cycles for each application. The name of setting is expressed as *Dn/Rm*, *Dn*, or *Rm*, where n levels of the LPC from the root are set to write-back and the other $m(= 24 - n)$ levels are write-through. *R24* is identical to the Reuse scheme. Graph (b) summarizes the difference of the number of DRAM accesses (left) and the difference of the miss rate in row buffers (right). In similar to Table 3, the average of each of them is only shown because both of them did not vary with application.

From Figure 8, the peak of the reduction of execution time was found at *D8/R16* for the all applications. The reduction of DRAM accesses per ORAM access almost reached at the upper limit at *D8/R16*, while the row buffer miss steadily rose by increasing the threshold. Thus, the Reuse and Delay schemes were balanced at this point.

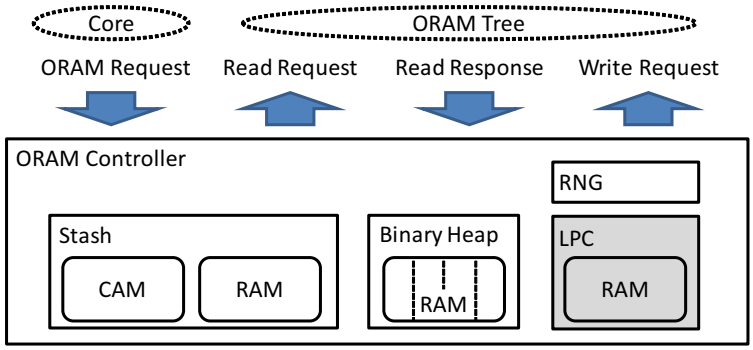


Figure 9: Block diagram of our prototype of ORAM controller.

6 Hardware Implementation

6.1 A Prototype of ORAM Controller

Figure 9 depicts our prototype of ORAM controller, which implements minimal elements of an ORAM controller. Blue arrows represent stream input and output. It does not include the position map: the leaf ID of the requested block is included in an ORAM request stream. It does not include the ORAM tree: input from and output to the ORAM tree are treated as three kinds of stream I/O i.e. read request, read response, and write request streams. The stash only holds tags of blocks. The stash is composed of a CAM (content-addressable memory) for looking up the stash for the requested block and a RAM for storing tag, such as corresponding leaf ID, for each block. A binary heap coordinates with the stash. It reorders the stash to find blocks to be written back. The block to be written back next always comes to the head of the binary heap. In addition, a random number generator (RNG) is required for remapping blocks.

6.2 Modification with Last Path Caching

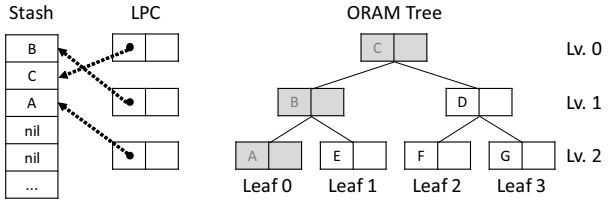
To implement the last path caching, an LPC must be added to ORAM controller, which is shown in gray in Figure 9. However, it only has to memorize the indices of the stash of the lastly written blocks, which correspond to less than 1k bits of RAM.

Figure 10 demonstrates a pointer array implementation of the last path caching. The point of this modification is that the blocks pointed by the LPC i.e. blocks written back in the last path-write-back stage remain valid in the stash. A non-overlapped part of such blocks is removed from the stash in the next path-write-back stage. Figure 10 (a) depicts the ORAM states with the pointer array implementation after writing $\mathcal{P}(0)$ back. In contrast to Figure 4 (a), the stash still holds the blocks A, B, and C, each of which is pointed by the corresponding entry of the LPC. When the block E is requested in Figure 10 (b), blocks in the overlapped part between the last path ($\mathcal{P}(0)$) and the current path ($\mathcal{P}(1)$) has to be read to the stash again. It is simply done by leaving them valid in the stash. In Figure 10 (c), since the block A belongs to the non-overlapped part, the block A is invalidated on writing back $\mathcal{P}(1)$. If the leaf level of LPC is set to write-back (i.e. the Delay scheme), the block A is written to the ORAM tree.

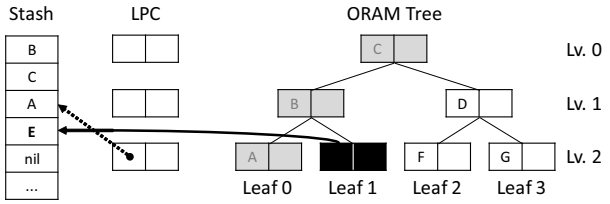
An advantage of this implementation is that a search for the LPC can be integrated in the search for the stash in the first step of the ORAM access, Since blocks pointed by the LPC are valid in the stash, they can also be found by looking up the stash, which is done by a CAM. Thus no extra clock cycles are required to determine whether a requested block resides in the LPC.

The path-read step of ORAM controller has to be modified for an overlapped part. Since it is already in the stash, read accesses to the ORAM tree are omitted. Instead, if real blocks are found in the overlapped part by reading the LPC, they must be registered to the binary heap.

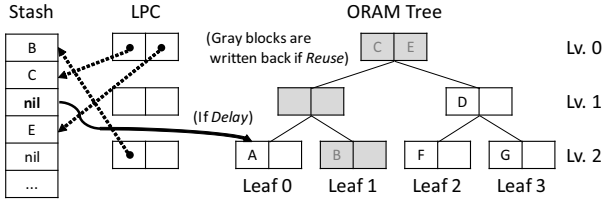
Modifications to the path-write-back step are threefold. First, if the current level is in the non-



(a) Last Path Cache can be implemented as a pointer array.



(b) Reading $\mathcal{P}(1)$ for the block E.



(c) Writing $\mathcal{P}(1)$ back for the block E.

Figure 10: Pointer array implementation of the last path caching.

overlapped part and belongs to the Delay scheme, blocks in the LPC are written to the ORAM tree, instead of the head of the binary heap. Second, the blocks in the non-overlapped part of the LPC are removed from the stash, instead of the head of the binary heap. Third, the index of the head of the binary heap is written to the LPC if that block can be written in the current level. If not, an invalid index is written to the LPC.

6.3 Implementation Results

This section describes the operating frequency and the logic scale of the prototyped ORAM controller. Xilinx AC701 evaluation board with an Artix-7 XC7A200T FPGA was adopted for this evaluation. Xilinx Vivado 2016.3 was used for logic synthesis and implementation. Synthesis and implementation options were default, with an exception of additional option `-mode out_of_context`, which prevent I/O ports to be placed to specific position. The amount of hardware was evaluated with the numbers of LUTs, flip-flops (FFs), and block RAMs (BRAMs). The maximum frequency f_{max} was calculated from the target frequency f_{targ} and the worst negative slack (WNS) reported from the tools. The target frequency was adjusted at 2-MHz intervals in order not to cause timing errors. CAM in the stash is implemented with a Xilinx IP [18]. RAM in the stash is implemented using Block RAMs, while the other RAMs (i.e. the binary heap and the LPC) uses LUTs because they are too small. We adopted an Xorshift random number generator [11] as an RNG.

We evaluated the original Path ORAM (*Normal*), the Delay/Reuse Hybrid scheme of the last path caching (*LPC*), and Fork Path ORAM (*Fork*). The height of the ORAM tree (L) was 14, 17, 20, and 23. The number of slots per bucket (Z) was set to 4. The size of the stash is set to 256.

Table 4 summarizes the hardware implementation results. The number of LUTs of *LPC* was 1.4%–7.8% smaller than *Fork*. The number of bits for the RAM in the LPC is calculated as a product of $(L + 1)$, Z, and logarithm of the stash size (to base 2). In this evaluation setting, it becomes up to 768 bits, which is equivalent to only 12 LUTs. Moreover, as we have explained

Table 4: Result of hardware implementation.

	$L = 14$			$L = 17$		
	Normal	LPC	Fork	Normal	LPC	Fork
LUT	1,908	2,025	2,054	1,953	1,974	2,131
FF	794	875	829	846	938	886
BRAM	16.5	16.5	16.5	16.5	16.5	16.5
F_{max} [MHz]	126.1	108.0	123.2	120.3	104.0	124.5
	$L = 20$			$L = 23$		
	Normal	LPC	Fork	Normal	LPC	Fork
LUT	2,125	2,147	2,329	2,418	2,301	2,489
FF	902	1,001	947	951	1,064	1,003
BRAM	25.0	25.0	25.0	25.0	25.0	25.0
F_{max} [MHz]	127.1	109.3	116.5	120.0	107.4	118.5

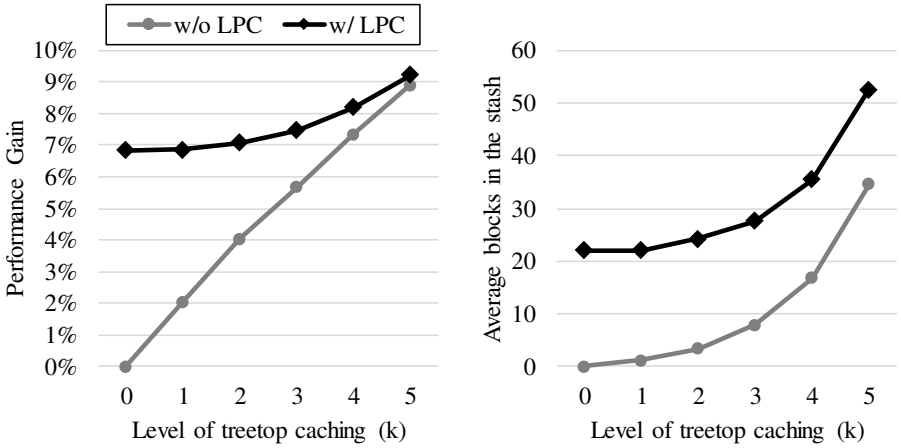


Figure 11: Overview of the evaluation results with the treetop caching.

in Section 6.2, the modification of the ORAM controller with the last path caching is minimal, which could keep the state machine of the controller simple. The increase of flip-flops probably came from temporal registers to check if the current level is in the overlapped part or not. The number of block RAMs was not changed because it came only from the stash in this prototype. The maximum frequency of *LPC* was 6.2%–16.5% lower than *Fork*. It might be because reading the LPC in zero-cycle caused longer combinational paths.

We had an unintended result seen in $L = 23$: the number of LUTs of *LPC* was smaller than *Normal*, the original Path ORAM. Though it might be due to optimization in logic synthesis, the detailed analysis is left as future work.

7 Related Work

After the algorithm [14], the first implementation [10], and an exploration of design space [13] of Path ORAM were presented, many optimization techniques and ORAM architectures similar to Path ORAM have been proposed, including Fork Path ORAM [20] explained in Section 3.

The PHANTOM implementation [10] applied another caching technique, called treetop caching, to the ORAM tree. It caches all blocks on a few levels of the tree from the root. It is achieved by skipping access to these levels and leaving blocks, which would be moved to the tree, be untouched in the stash [9]. Our last path caching is also implemented by leaving some blocks in the stash.

In this sense, our method is similar to the treetop caching. Since nodes near the root are the most frequently accessed, the treetop caching leverages spatial locality of the ORAM tree, while our last path caching utilizes temporal locality. Though the effect of these two caching techniques are partially overlapped, using both of them gives higher performance. Figure 11 is the overview of the evaluation results of combination of both caching with a similar environment as shown in Section 5.1. The left and right graphs show the average performance gain and the average number of blocks in the stash after ORAM access, respectively. For example, as we have evaluated in Section 5.3, the average performance gain of *D/R+Q8* was 6.9%, while 3-level treetop caching increased the performance by 5.7%. The combination of these two caches achieved 7.5% of the performance gain. Due to the overlap of the two types of locality, the advantage of the last path caching was reduced as the number of levels k increased. However, the average number of blocks in the stash also increased with higher k, which put pressure on the size of the stash. It was almost proportional to the number of nodes to be cached, which was $(L+1)$ in the last path caching and 2^k-1 in the treetop caching. Therefore, the combination to the treetop caching should be discussed as a trade-off between the performance and the pressure on the size of the stash.

Fork Path ORAM was proposed with another caching technique called Merging Aware Cache (MAC) [20], which may solve an overlap of effect with the treetop caching. Unlike the last path caching and the treetop caching, MAC is an actual cache organization for nodes in the intermediate levels, say, level m_1 to level m_2. Each level r of the MAC has 2^{r-m_1+1} entries to hold recently accessed blocks of that level. However, we currently do not consider the MAC for two reasons. The first reason comes from a limitation in our simulation environment. As we have explained in Section 5.1, we limit the movement in a reordering of ORAM requests [20] to one-way in order to prevent starvation. This is because we did not prevent the performance loss due to starvation using an age-based method presented in [20]. However, this modification makes MAC useless: in a round-robin, the lastly accessed blocks are least likely to be reused. The second reason is that the MAC is too large to compare with the other caching methods. For fair comparison, if we supposed a large cache to the ORAM tree, we would have to evaluate the case of simply enlarging the last level cache in the processor. A detailed analysis with a larger cache, including the MAC, is left as future work.

Ring ORAM [12] is an ORAM architecture based on Path ORAM. In the original Path ORAM, if a node is included in the accessed path, all blocks in the node are read out to the stash. Ring ORAM limits the number of blocks read by an ORAM access to only one per node. As a result, the total number of read blocks becomes unproportional to Z, the number of blocks for each node. However, it may reduce the efficiency in capacity because of an increased number of dummy blocks. Although it was reported that it did not affect in a practical use [12], the detail has not been reported. The idea of separate treatment of the path-read and the path-write-back was also seen in RAW ORAM [3]; its goal was to reduce the cost of encryption, which is not related to our research.

When the position map is too large and the recursive approach is applied as explained in Section II.A, Freecursive ORAM [2] is effective. It introduces a PosMap Lookaside Buffer (PLB), a cache to the position map, which greatly reduces the ORAM accesses caused by looking up the position map. Although the recursive approach is not currently considered in our research, the proposed schemes do not interfere the use of the PLB.

Prefetch is often useful when a running program has high spatial locality. In Path ORAM, blocks can be prefetched by making multiple blocks always mapped to the same path. This technique is called super block [13]. PrORAM [19] extends it to a dynamic approach: it merges adjacent blocks into a super block or unmerges super blocks on the fly. It showed higher performance than the static approach by unmerging useless super blocks. These approaches are also orthogonal to the proposed schemes.

8 Conclusion

This paper proposed a technique to remove the redundant memory accesses in Path ORAM, which was simpler than existing Fork Path ORAM and was determinate in the access pattern derived by ORAM. Our evaluation showed that the performance was comparable to the existing method.

The advantage of the reduced dummy accesses was also observed when ORAM requests were not frequently arrived. A prototyped hardware implementation was also presented along with its evaluation. The number of LUTs used was 1.4%–7.8% smaller than the existing method.

Our future work includes a detailed analysis on the applicability of the proposed technique, especially with a larger cache to the ORAM tree.

Acknowledgement

This study was partially supported by JSPS Grants-in-Aid for Scientific Research (KAKENHI), Grant Number 26870278 and 16K00072.

References

[1] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. USIMM: the Utah SImulated Memory Module. Technical Report UUCS-12-002, University of Utah, 2012.

[2] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 103–116, 2015.

[3] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. A Low-Latency, Low-Area Hardware Oblivious RAM Controller. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 215–222, 2015.

[4] Naoki Fujieda, Ryo Yamauchi, and Shuichi Ichikawa. Last Path Caching: A Simple Way to Remove Redundant Memory Accesses of Path ORAM. In *Proceedings of the 4th International Symposium on Computing and Networking*, number 347–353, 2016.

[5] Naoki Fujieda, Ryo Yamauchi, and Shuichi Ichikawa. Preliminary study on the reduction of bandwidth overhead for Path ORAM. In *IPSJ SIG Technical Report 2016-ARC-219*, pages 17:1–17:6, 2016. (in Japanese).

[6] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 182–194, 1987.

[7] Michael Henson and Stephen Taylor. Memory Encryption: A Survey of Existing Techniques. *ACM Computing Surveys*, 46(4):53:1–53:26, 2014.

[8] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantacioglu. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, 2012.

[9] Martin Maas. PHANTOM: Practical Oblivious Computation in a Secure Processor. Technical Report UCB/EECS-2014-89, University of California at Berkeley, 2014.

[10] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 311–324, 2013.

[11] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[12] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *Proceedings of 24th USENIX Security Symposium*, pages 415–430, 2015.

[13] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 571–582, 2013.

[14] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 299–310, 2013.

[15] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171, 2003.

[16] The Journal of Instruction Level Parallerism. 3rd JILP Workshop on Computer Architecture Competitions (JWAC-3): Memory Scheduling Championship (MSC). `http://www.cs.utah.edu/~rajeev/jwac12/`.

[17] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems*, pages 168–177, 2000.

[18] Xilinx Inc. *Parameterizable Content-Addressable Memory*, 2011. Application Note XAPP1151 (v1.0).

[19] Xiangyao Yu, Syed Kamran Haider, Ling Ren, Christopher Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. PrORAM: Dynamic Prefetcher for Oblivious RAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 616–628, 2015.

[20] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. Fork Path: Improving Efficiency of ORAM by Removing Redundant Memory Accesses. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 102–114, 2015.