Fork Path: Batching ORAM Requests to Remove Redundant Memory Accesses

Jingchen Zhu, *Student Member, IEEE*, Guangyu Sun, *Member, IEEE*, Xian Zhang, *Student Member, IEEE*, Chao Zhang, *Student Member, IEEE*, Weiqi Zhang, *Student Member, IEEE*, Yun Liang, *Member, IEEE*, Tao Wang, *Member, IEEE*, Yiran Chen, *Fellow, IEEE*, and Jia Di, *Senior Member, IEEE*

***Abstract*—Outsourcing data to a third-party cloud provider has become quite common with the increasing use of cloud computing. This brings convenience, as well as the concern for data security and privacy. It is believed that data encryption alone is often not enough to protect users' privacy from the cloud provider. According to previous work, the sequence of storage locations accessed by the client can leak up to 90% of the sensitive information, even with data encrypted. In this context, Oblivious RAM (ORAM) is proposed. ORAM algorithms allow the client to hide its access pattern from the service provider while introducing a lot of extra operations. Among all the prototypes, Path ORAM is one of the most promising designs. However, there are still redundant memory accesses that can be removed without harming the security of traditional ORAM as we observed. We came up with three optimization techniques, including path merging, ORAM request scheduling, and merging aware caching. We also propose a prefetching technique to further decreasing the access overhead. Moreover, we also illustrate the compatibility of Fork Path and some state-of-the-art Path ORAM optimizations. Compared to traditional Path ORAM approaches, our Fork Path ORAM can reduce overall performance overhead and power consumption of memory system by 65% and 44%, while the design overhead is trivial.**

***Index Terms*—Access merging, Oblivious RAM (ORAM), request scheduling.**

I. INTRODUCTION

NOWADAYS, outsourced storage applications has become an important part of the cloud computing service. Since most of the time cloud provider is not trusted, the concern for data security and client privacy is raised. Recent researches have pointed out that a large amount of private information can be leaked through the memory access pattern [1]–[3], which makes merely data-encrypting insufficient. To overcome this privacy leak problem, Oblivious RAM (ORAM) that was proposed thirty years ago [4], [5] has attracted lots of attention recently.

ORAM is a cryptographic primitive, which allows a client to hide its access pattern to the remote storage by reshuffling and re-encrypting data every time a memory block is accessed [6]–[8]. Using ORAM, any memory access pattern is computationally indistinguishable from others of the same length [9], [10]. However, the overhead of ORAM memory access is always unacceptable. Obviously, traditional ORAM implementation requires a considerable amount of extra memory accesses. According to previous work, ORAM introduces $10\times$–$100\times$ more memory accesses compared to the unprotected baseline [7], [9], [11], leading to significant memory latency. For those memory-intensive applications, this could lead to up to 10x performance degradation [7], [9], [10]. With more and more secure processors using chip-multiprocessor and out-of-order pipelining architectures to achieve higher memory bandwidth, it is conceivable that the performance of traditional ORAM would become a bottleneck limiting its large-scale application [12]–[15].

To overcome this limitation, various approaches have been proposed to increase the efficiency of ORAM [6], [8], [16]–[18]. Among all, an ORAM scheme called Path ORAM [18] stands out with its simplicity and high efficiency. The external memory is structured as a binary tree consisting of buckets as nodes, while each of the buckets contains several blocks with data encrypted. Any access to a specific block results in a full-path visit from the root to one leaf, which reduces the overhead of extra accesses effectively. It was proved that Path ORAM provides the same security as traditional ORAM. Moreover, several follow-up techniques have been proposed these years [7], [9], [10], [19]–[22], making Path ORAM one of the most efficient approaches. However, for many memory-intensive applications today, the performance of Path ORAM is still not in a practical and acceptable range.

We find that there is a potential to further improve the performance of Path ORAM by batching ORAM requests, which can remove some of the redundant memory accesses. For a sequence of memory requests given by the client, Path ORAM results in a block visit sequence, which contains a lot of read and write back operations. Among all these operations

Manuscript received May 28, 2019; revised September 17, 2019; accepted September 23, 2019. Date of publication October 22, 2019; date of current version September 18, 2020. This work was supported in part by the National Key Research and Development Project of China under Grant 2018YFB1003304, and in part by the National Natural Science Foundation of China under Grant 61572045. This article was recommended by Associate Editor Z. Shao. *(Corresponding author: Guangyu Sun.)*

J. Zhu and X. Zhang are with CECA, Peking University, Beijing 100871, China (e-mail: zjc990112@pku.edu.cn; zhang.xian@pku.edu.cn).

G. Sun, Y. Liang, and T. Wang are with CECA, Peking University, Beijing 100871, China, and also with the Advanced Institute of Information Technology, Peking University, Hangzhou 311200, China (e-mail: gsun@pku.edu.cn; ericlyun@pku.edu.cn; wangtao@pku.edu.cn).

C. Zhang and W. Zhang were with CECA, Peking University, Beijing 100871, China (e-mail: zhang.chao@pku.edu.cn; zhangweiqi@pku.edu.cn).

Y. Chen is with the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708 USA (e-mail: yiran.chen@duke.edu).

J. Di is with the Computer Science and Computer Engineering Department, University of Arkansas, Fayetteville, AR 72701 USA (e-mail: jdi@uark.edu).

Digital Object Identifier 10.1109/TCAD.2019.2948914

0278-0070 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

between the external memory and private storage, some operations are able to be removed without harming the security of ORAM design as we proved.

In this article, we propose a Fork Path ORAM scheme to remove redundant memory accesses by batching ORAM requests, improving performance while maintaining security.

Fork Path ORAM makes the following contributions.

1) Unlike Path ORAM which focuses on independent operations, we consider optimization techniques from the perspective of a sequence of memory accesses. Since the access to external memory is always known by the third-party service provider, we prove that this leaks no more information than Path ORAM does.
2) We propose a path merging technique to avoid redundant memory accesses, and also consider the extra dummy operations this may introduce. To reduce the extra overhead, we propose a request scheduling technique.
3) We observe that treetop caching applied in traditional Path ORAM became inefficient after path merging, thus we present a merging-aware caching (MAC) strategy to improve the performance.
4) We propose a prefetching technique combining with path merging and request scheduling to further reduce the number of ORAM requests.
5) We specify the modification of Fork Path to accommodate state-of-the-art variants of Path ORAMs and illustrate that our optimizations are applicable to other ORAMs as well.
6) We propose a detailed architecture of the ORAM controller. We present the theoretical performance and matching the experimental results to demonstrate the efficiency of our optimizations.

The rest of this article is organized as follows. Section II introduces the thread model and the basic ORAM implementation. We also provided some state-of-the-art Path ORAM schemes including Tiny ORAM, as the baseline of this article. Section III introduced our Fork Path ORAM in detail, including path merging, request scheduling, request prefetching, and MAC. Section IV presents a detailed architecture of ORAM controller. Section V gives comprehensive evaluations of our design and compares with the baseline Path ORAM implementions. Section VI introduces the related works on ORAM research, and Section VII makes a conclusion.

II. Background

A. Threat Model

As a security-oriented design, a threat model is necessary to be presented first when we consider ORAM implementations. The threat model we used here is similar to those proposed in previous works [7], [9]. We assume that users outsource data and private programs to the remote servers that are physically accessed by third-party service providers. With the data stored on the external storage, the private processor needs to access the remote memory, read data to local memory, or write back data to external memory during the process of a program. The local memory inside the processor is trusted and invisible to the service providers, while the access of external memory is easy to be detected [7], [9], [10]. Since the external memory is untrusted, without protection, a lot of private information can be leaked through the access, including data stored on the cloud memory, the addresses on the bus, and private informations about the details of the program.

Traditional encryption schemes can provide confidentiality [23]–[27], but the access pattern can not be hidden. Just by the access pattern on the memory bus, sensitive information, such as the encryption type or even secret keys can still be learned by the attacker [1]–[3]. The goal of ORAM is to completely hide the access pattern from the server.

B. Security Definitions

In this article, we adopt a standard definition of ORAM as proved in previous work. We consider an ORAM design to be secure when the server learns nothing about the access pattern. An ORAM design is secure in the following rule [8], [10], [18].

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.

C. ORAM Basics

ORAM hides the access patterns from the server completely. In our definition above, the $\overrightarrow{a}$ represents the sequence of memory requests from the program, instead of the ORAM requests transposed by the ORAM controller. For every memory request, we can come up with an extreme idea that we read all data into our local processor, read, and write back with data reshuffled. This promises that no information about the accessed data is leaked, though impractical with huge overhead. Moreover, since data caching is normally employed in secure processors, ORAM($\overrightarrow{a}$) normally refers to cache misses of last level cache (LLC). This leads to a problem that the length of ORAM($\overrightarrow{a}$) may indicate the number of cache hits. Previous works have proved that information leaks logarithmically with the increasing length of ORAM($\overrightarrow{a}$). To overcome this leakage, a nonstop stream of accesses can be used. Thus, memory requests to the external memory reflects no information about whether there are LLC misses or not. In other words, from the perspective of the service provider, read or write operations are indistinguishable from random requests.

D. Path ORAM

In this part, we introduce a state-of-the-art ORAM design, Path ORAM, and the memory access flow under this protection, which we use as the baseline of our design. Path ORAM is to date one of the most practical ORAM construction under small client storage [9], [10], and has been employed in some current secure processors [7], [9], [10], [28]. It is accepted that Path ORAM is a promising and efficient ORAM implementation that reduces the overhead of ORAM to a logarithmically level. In Fig. 1, we present an overview of Path ORAM architecture and the data flow of a memory access operation.

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

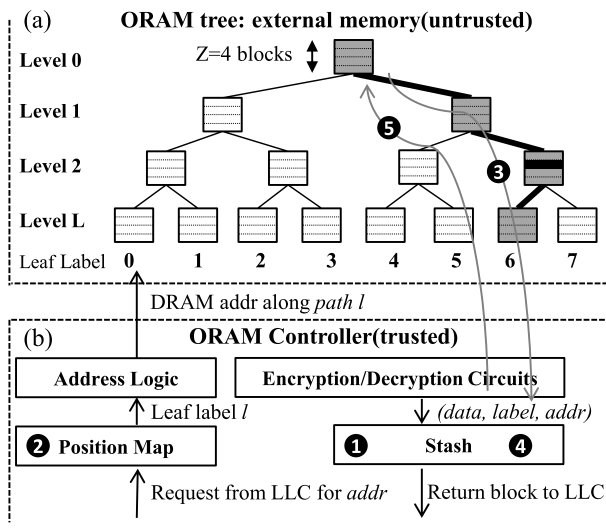


Fig. 1. Illustration of Path ORAM architecture that consists of (a) ORAM tree and (b) ORAM controller ($L = 3$ and $Z = 4$).

This Path Oram architecture consists of two parts. The upper half indicates untrusted external memory. In this design, the external memory is organized as a binary tree [9], [10]. In fact, the tree does not have to necessarily be a binary tree, but we use a binary tree to make our description simple. The lower half indicates the trusted ORAM controller. The on-chip ORAM controller is designed to be a part of the secure processor, which has an interface for LLC as we mentioned.

The ORAM tree has $L+1$ levels, ranging from level 0 (root) to level L (leaf). Each of the nodes in the tree is called a bucket, and every bucket is internally divided into a fixed number(Z) of blocks, which is the smallest unit of data access. The capacity of external storage is always larger than the valid data in order to provide storage for dummy blocks. Encryption technology is applied to external storage. Both data blocks and dummy blocks are encrypted and stored in the external memory, thus any two blocks are indistinguishable even their plain data are the same. Another important concept in Path ORAM is the path, since every memory request would result in a path-based storage access to achieve security. A path means a set of buckets from the root to one leaf node in the binary tree. Once the structure of the tree is established, the length of a path is fixed. In Fig. 1, we highlight path-l in gray as an example.

The ORAM controller has a local storage consisting of two parts: 1) a position map and 2) a stash. During the course of the algorithm, the client locally stores a small number of blocks in a local memory structure named stash. The storage of the client processor is always supposed to be trusted, so the data in stash does not need to be encrypted. When the required data is read in or written back, a encryption/decryption logic is needed. The position map is a lookup table recording the run-time mapping relationship between blocks and leaf nodes.

In Path ORAM, we do not need to distinguish between different blocks on the same path. When we need to access a block, we access the whole path corresponding to the leaf node, read it into stash, and searched for the block according to the decrypted index. When a block is accessed, it is remapped to a new leaf label, making one of the copies invalid. Path ORAM design holds the following invariant [7], [18]: a data block mapped to leaf label l must be either in the stash or path l.

To summarize, for every memory request denoted as $(addr, op, data)$, Path ORAM works in the following steps [10].

1) *Step 1:* The ORAM controller receives a memory request and searches stash first. If the required data block can be found in stash, return to LLC immediately; else goes to step 2.
2) *Step 2:* The required data is stored in the external memory. ORAM controller searches the position map by indexing with *addr* and gets the leaf lable(l) to be accessed. Then the target block is remapped to a new leaf label l' and the position map is updated.
3) *Step 3:* The whole path containing the leaf node, including the target block is read from the external memory, decrypted, and stored into the stash. The required data blocks are forwarded to LLC.
4) *Step 4:* The required data block has two copies at this time: one in stash, one in external memory. Since the block is remapped to a new leaf lable, the copy in stash is updated, and the copy in external storage becomes out-of-date.
5) *Step 5:* The buckets on path-l needs to be refilled with blocks. Every block in stash is scanned and the blocks that can be written back to the path are refilled to the memory path as many as possible. If there are still empty blocks, dummy blocks are inserted.

In a design of Path ORAM, the memory allocation of the secure processor needs to be considered rigorously. On the one hand, stash needs to be big enough to hold the data in a path, and also the remaining blocks in previous accesses. A proper stash size(C) should be set to mitigate the possibility of stash overflow. As an example, it is discussed that when the utilization of an 8-GB DRAM is 50% while $C \geq 200$ and $Z \geq 4$, the possibility of stash overflow is negligible [9], [10], [18]. On the other hand, when the size of data blocks is relatively small or the number of blocks keeps growing, the position map would be too large to be stored on-chip. Thus, hierarchical ORAM is proposed [7], [18]. Hierarchical ORAM puts the position map in external memory and protects the PosMap blocks basing on Path ORAM. Hierarchical ORAM keeps a separate PosMap ORAM, which introduces extra memory accesses. Unified ORAM [9], [19], [29] is proposed to solve this problem by keeping PosMap blocks and data blocks in the same address space. In the rest of this article, the unified hierarchical Path ORAM is used as our baseline for discussions and denoted as Path ORAM for simplicity.

A nonstop stream of accesses is also used in this design to protect the timing channel of Path ORAM [10], [30]. As is shown in Fig. 2(a), each ORAM request consists of a read phase and a write phase. Between the two memory operations and every two of the requests, a fixed idle phase is inserted. By launching ORAM requests in a constant rate continuously, though we make a loss of the performance, the fixed response time leaks no information about the cache. When there is

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

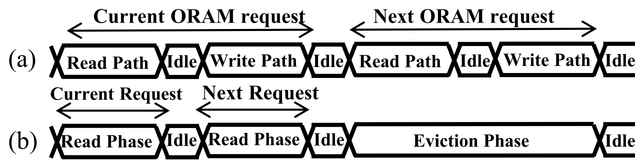


Fig. 2. Timing diagram of (a) Path ORAM and (b) Tiny ORAM with $A = 2$.

no LLC miss, dummy requests are launched to maintain the instruction cycle.

E. Tiny ORAM

As we mentioned above, traditional Path ORAM designs are divided into a form of alternating read and write phase in the time domain. A state-of-the-art variant of Path ORAM is a type of multiread-single-eviction ORAMs [8], which is also called MRSE ORAMs, including Tiny ORAM [19] and Ring ORAM [29]. For simplicity, we use Tiny ORAM as an example to describe the features of multiread-single-eviction (MRSE) ORAMs. Other MRSE ORAMs are quite similar.

MRSE ORAMs, as the name suggests, have a different performance than traditional ORAMs in timing. As shown in Fig. 2(b), there are two phases in these ORAMs: the read phase and the eviction phase. The function of the read phase is only to fetch the intended block by a path read. And the eviction phase is only used to evict blocks in the stash by a path read and a path write following a reverse lexicographical order [19], [29]. An eviction phase always occurs after a fixed number of read phases, called the eviction rate in this article. Compared to Path ORAMs, MRSE ORAMs can theoretically achieve a lower access overhead and higher performance. In the rest of this article, a Tiny ORAM combined the unified ORAM is used as a representative of Path ORAM variants for discussions and denoted as Tiny ORAM for simplicity.

III. Fork Path ORAM Scheme

A. Motivation

Path ORAM has some redundant operations. Fig. 3 shows an example of Path ORAM requests. In this example, we simply put one block in a bucket and the letter in each bucket represents the data stored in it. Consider two ORAM paths with leaf label 1 and label 3 accessed consecutively, based on the steps we introduced above, the memory access sequence in traditional Path ORAM design is shown in Fig. 3(a)–(d). When a block with leaf label 1 is accessed, all buckets along the path from root to leaf node 1 (in Fig. 3 A, B, C, D) are decrypted and loaded into stash. Then after the required data blocks are forwarded to the LLC, the path is refilled with write-back data (A', B', C', D'). For the next memory request to path 3, (A', B', E, F) are loaded into stash and refilled with (A," B," E', F').

Traditional Path ORAM focus on independent memory operations and reduced the overhead of each operation. However, from the perspective of a sequence of memory accesses, some of the accesses are redundant to be removed. In this example, we observe that for the overlapped part of

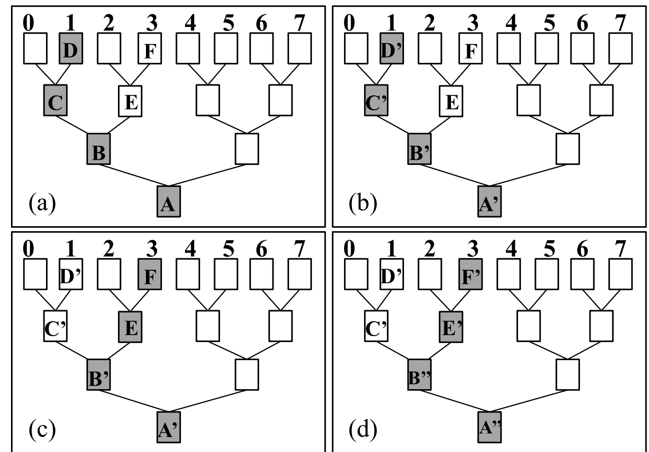


Fig. 3. Read/write phases for two adjacent requests accessing (a) and (b) path-1 and (c) and (d) path-3.

two paths, bucket A' and B' are written back to the external memory during the Write Path phase of the first request, while loaded into stash by the second memory request intactly. Notice that this part of the information is visible to the adversary. The data is encrypted when uploading to the cloud, and decrypted to be loaded into stash. This operation is considered to be redundant and can be removed without harming the security provided by traditional Path ORAM.

To conclude, memory operations of writing and reading data in the overlapped region of consecutive ORAM requests are considered to be redundant and can be removed to further improve the efficiency of Path ORAM. We will introduce in detail a path merging technique in the following part to take advantage of this observation.

B. Path Merging

The basic idea of path merging is to avoid the operations on the overlapped part of contiguous Path ORAM requests. To achieve this, we propose a modified Path ORAM works in the following steps.

1) *Step 0:* For the first memory request after initialization, all buckets along the path are loaded into stash. Similarly, only the required data blocks are forwarded to LLC.
2) *Steps 1 and 2:* They are the same as previously mentioned in Section II-D.
3) *Step 3*: When we need to load data from the external memory, only the buckets that are not overlapped with the previous requests are loaded and stored in stash. In fact, the overlapped part of the path is supposed to be already in stash under the guarantee of our design.
4) *Step 4:* It is the same as previously mentioned in Section II-D.
5) *Step 5:* Only the buckets that are not overlapped with the pending requests needs to be refilled. The stash is scanned, proper data blocks or dummy blocks are written back to refill the buckets.
6) *Step 6:* When there is no pending request, a dummy request will be inserted to maintain the instruction cycle.

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

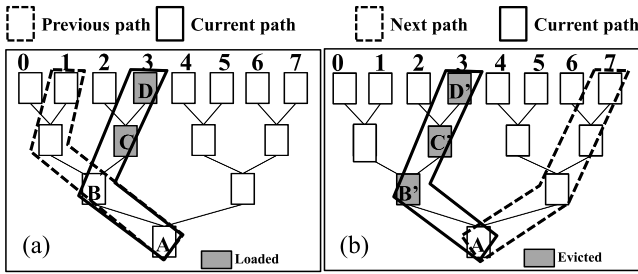


Fig. 4. Illustration of path merging. (a) Read phase of current path. (b) Write phase of current path.

The dummy request is similarly treated as memory requests in step 5.

We give an example of path merging in Fig. 4. In Fig. 4(a), buckets A and B are already in the previous path framed with a dotted line, so when we comes to step 3 of our current memory request, only C and D need to be loaded into stash in our design. Similarly, in Fig. 4(b), bucket A is not refilled since it is in the next path to be accessed.

By applying this path merging technique, the redundant operations between Path ORAM requests can be removed, and the required data blocks are accessed in the shape of a fork path. This optimization from the perspective of a sequence of memory accesses reduces the number of memory operations and shortens the response time of single request when the total number of memory requests is fixed, but also introduces extra dummy requests. We noticed the existence of this problem, which may offset our optimization and discussed this issue in the following section.

C. Dummy Label Replacing

Traditional Path ORAM provided protection on the timing channel by launching requests in a constant rate. After path merging, the time cost of single ORAM request is reduced. When there is no pending ORAM requests, a dummy ORAM request is inserted, thus compared with traditional Path ORAM, extra dummy requests are introduced when the memory request intensity from LLC is low. This is common when the secure processor is in-order and single-core, especially when hierarchical Path ORAM is employed.

It is simple to avoid the extra dummy requests by merely extending the idle phase inside a ORAM request. However, this prolongs the latency of data accesses. Actually, some of the dummy requests can be replaced by the incoming data request without being noticed by the external memory. By replacing dummy requests according to the following rules, the offset introduced by applying path merging can be reduced to some extent.

1) *Rule 1:* If the data request arrives when the dummy request is already launched, the dummy request can not be replaced.
2) *Rule 2:* If the dummy request is also in the waiting queue, but the bucket on the crossing point of the current path and the following data path is already refilled, the dummy request can not be replaced.

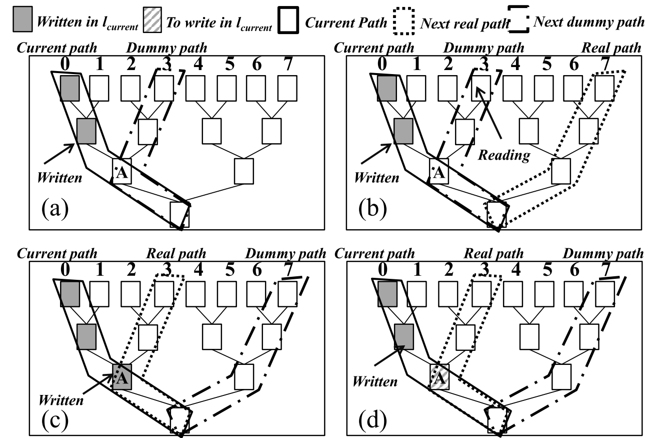


Fig. 5. Illustration of dummy request replacing. (a) Initial state. (b) Case-1. (c) Case-2. (d) Case-3.

3) *Rule 3:* In all remaining cases, the dummy request can be replaced by the following data request.

Fig. 5 provides examples of dummy label replacing under the above rules. In Fig. 5(a) we show the initial state of a sequence of requests that a current request is launched, followed by a inserted dummy request. In Fig. 5(b), the dummy request is already loading data from the external memory, so it is too late to relaunch a real data request in the constant rate. In Fig. 5(c), bucket A is refilled when the data request arrives. If we replace dummy request at this point in time, bucket A, as the overlapped part of the paths will not be loaded into stash according to our design, which may lead to unexpectable errors. While in Fig. 5(d), bucket A is not yet refilled. The operations of current request are consistent so far, whether the pending request is dummy request or data request. Under this circumstance, the dummy label can be replaced implicitly.

We can prove that no information is leaked through this dummy label replacing process. In Path ORAM, dummy requests and real data requests are indistinguishable to the external memory. The write process starts from the leaf node and descends toward the root in a path. Dummy requests are inserted to a request queue inside the secure processor, and not revealed until the previous requests finish the refill process. Therefore, dummy blocks can be replaced implicitly in time if the refill process performs normally as if the dummy request never exists.

D. ORAM Request Scheduling

A Path ORAM controller in the secure processor receives a memory request from LLC, transfers the request to ORAM requests, stores the ORAM requests in a request queue, and launch requests in a constant rate. Path merging provides us a method to avoid some overlapped operations between consecutive ORAM requests, thus we can reasonable schedule the requests in the queue to achieve higher overlap degree, which further reduces memory operations that interact with external storage. It is quite common in secure processors with multicore and/or using out-of-order pipelines that multiple pending requests exist. For those cases when the memory

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

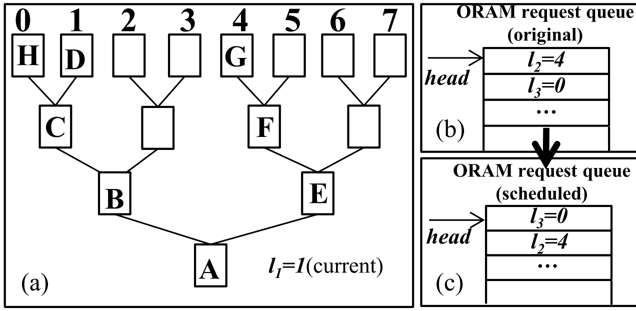


Fig. 6. Illustration of ORAM request scheduling. (a) ORAM tree. (b) Requests before scheduling. (c) Requests after scheduling.

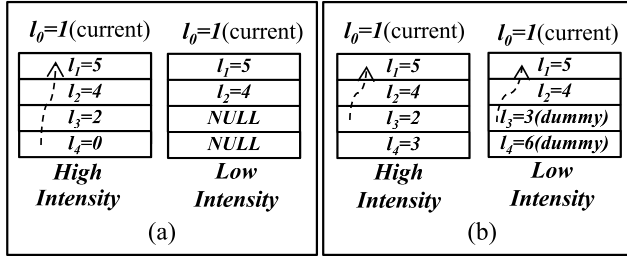


Fig. 7. (a) Scheduling among variable number of pending requests will leak information. (b) Dummy labels should be inserted if the queue is not full with data ORAM request.

access intensity is low, we provided a secure solution below as well.

In Fig. 6, we give a simple example of request scheduling. The current request accesses ORAM path with leaf label 1, and two pending requests accessing path-4 and path-0 are waiting in the request queue to be launched. As we can observe in Fig. 6(a), if we launch path-0 as our next request, we only need to refill bucket D and load bucket H to the stash. Compared with path-1 followed by path-4, extra operations on bucket B and C are removed, which leads to a better performance of our design. To conclude, we schedule the requests in the waiting queue and the request that has the highest overlap degree is selected as the next ORAM request for path merging.

By ORAM request scheduling, we can achieve a global optimal access sequence that requires minimal external operations. However, this scheduling process may lead to data hazards and fairness issues, which have been addressed in some previous works [10], [18]. In our Fork Path ORAM architecture, we further discussed and provided proper solution on this issue in Section IV.

When we schedule requests, it is worth mentioning that the number of requests in the waiting queue can leak some information about LLC. Obviously, the more ORAM requests pending in the queue, the higher efficiency path merging with scheduling can achieve. If we use a schedule strategy that depends on the number of requests in the queue, in some cases (e.g., we mentioned when the memory access intensity is low) private information can be leaked through this process, since the degree of path overlapping will reflect the intensity of LLC requests. Therefore we ensure the waiting request queue to be full all the time, with dummy requests filled when there is idle in the queue as shown in Fig. 7.

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
```

The insertion of dummy requests gives us the space to apply dummy request replacing. Similarly, the inserted dummy requests can be replaced by incoming requests in some cases. Algorithm 1 described the replacing rules in detail.

To avoid leaking information to the external memory, as we can see, dummy requests are possible to be launched before real requests anyway. A scheduling operation on the entire queue is applied, while for requests that has the same overlap degree with current request, real data requests has a higher priority to be launched than dummy requests. Through this, we can further reduce some of the operations by ORAM request scheduling in various possible situations.

E. Prefetch Request Insertion

As addressed in Section III-D, a critical drawback of request scheduling is the introduction of additional dummy requests since the waiting request queue needs to be fulfilled all along to avoid information leakage about LLC. In the evaluation section, we also observe that for some benchmarks, the imported dummy requests can severely offset the benefit of scheduling with a large label queue. Based on this, we propose to insert prefetching requests instead of dummy requests to mitigate the impact of dummy requests.

ORAM requests are supposed to be launched in a constant rate continuously, and when there is no pending ORAM request, prefetched requests would be launched, right in the constant rate as real ORAM requests, and sent to the label queue. Compared with dummy requests, no extra CPU operations are added and this overhead is necessary for secure consideration. In our design, we implement a straight-forward prefetching scheme which takes advantages of spatial locality. By analyzing the address queue, our prefetcher will add CPU requests according to the following three rules.

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

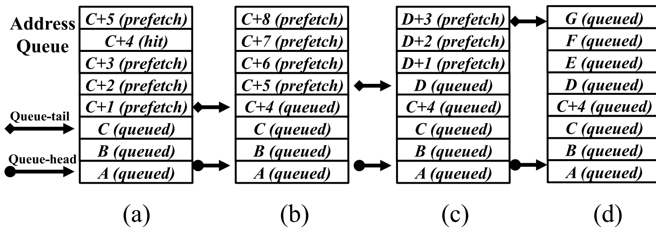


Fig. 8. Illustration for prefetching (a) prefetched requests based on address C (b) address C+4 is hit and addresses before are flushed (c) addresses after C+ 4 are flushed and (d) prefetched requests are replaced by incoming requests.

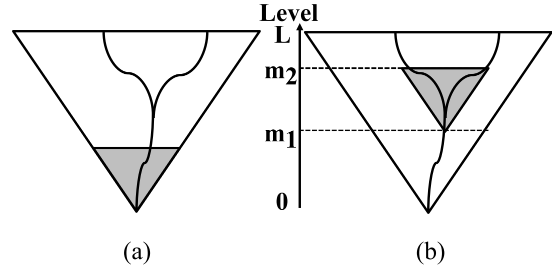


Fig. 9. (a) Treetop caching versus (b) merging aware caching.

1) The prefetcher inserts requests accessing the consecutive addresses of the last queued request.
2) Once an LLC request arrives and is found previously queued ("hit"), the inserted addresses ahead of this request is flushed; If a request miss occurs, all the inserted addresses are flushed and next insertions are based on the new address.
3) Any prefetched request can be replaced by the incoming LLC request.

To better explain the rules of our prefetching design, several cases are presented below in Fig. 8. As illustrated in Fig. 8(a), A, B, C are the queued addresses and prefetched addresses are always based on the last queued address (i.e., "C"). Data blocks near c are prefetched into the queue to promise the queue is full. If we suppose that $C + 4$ is hit, prefetched addresses before $C + 4$ are flushed to ensure that the real ORAM request has a higher priority to be launched than prefetched ones while $C + 5$ and consecutive addresses are inserted [Fig. 8(b)]. In Fig. 8(c), the advent of Address D results in the flush of $C + 5$ to $C + 8$. And the incoming requests always have a higher priority that they can replace the prefetched requests, as shown in Fig. 8(d). In other words, prefetched requests in the address queue are always preparing to be replaced by incoming real requests, or updated by a better prefetching choice.

Previous work [11] has also proposed a specific prefetching technique that maps several consecutive addresses to a same path. Therefore, one ORAM fetch can potentially load several future addresses following the spatial locality of programs. However, their motivation of prefetching is different from ours. The goal of their mapping process is to reduce the data requests [11] while in our scheme we use the prefetching technique to reduce dummy requests. Besides, the join operation in will increase the possibility of stash overflow while our prefetching has no impact on this.

F. Merging-Aware Caching

For current secure processors, on-chip data caching is applied to reduce the overhead of ORAM [10]. The frequently accessed data blocks are cached in the memory of ORAM controller, which reduces the response time for some of the requests using memory locality. Traditional Path ORAM always adopt treetop caching as the baseline of design, in which buckets in the levels close to the root of ORAM tree are cached in private storage. Due to the access mode of Path ORAM, it is obvious that the nodes at lower levels are more frequently accessed than those at high levels. Thus, treetop caching can achieve a good performance. However, in our design using path merging, some of the accesses to the buckets close to the root are removed and the frequency of visits to them is greatly reduced. In our design, we apply MAC to maximize the efficiency of on-chip cache.

In traditional Path ORAM, data blocks are accessed in units of paths. Each node of the path is read and write back in one visit. While in Fork Path, with path merging technique, buckets are accessed in a fork style just like its name. The overlapped parts of consecutive paths are merged, which makes data blocks on the tines of the fork to be accessed more frequently than those on the handle of the fork. The cache should focus on buckets that are always accessed, rather than those considered to be always in stash. Obviously, the attention of treetop caching is on the latter. For a Fork Path design, if the average overlapped path length is assumed as $\text{len}_{\text{overlap}}$, it is almost useless to cache data in the levels lower than $\text{len}_{\text{overlap}}$. MAC allows us to cache the blocks higher than $\text{len}_{\text{overlap}}$ to achieve a better performance.

Fig. 9 is a more intuitive comparison of the difference between the two caching policies. The entire triangle represents the external storage organized into a binary tree, and a fork path is shown as the access mode applying path merging. Fig. 9(a) represents the traditional tree-top caching, and Fig. 9(b) represents merging aware caching from the level higher than m_1. When the size of cache is fixed, represented as a shaded triangle in the figure, merging aware caching covers more of the required blocks, which optimizes the request latency. In the actual design, m1 is always set to $\text{len}_{\text{overlap}} + 1$, and m_2 depends on the on-chip cache size. Moreover, 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. An LRU replacement policy is very suitable in a sense. Details on the cache design will be presented later in Section IV.

G. Tiny ORAM With Fork Path

The combination of Tiny ORAM and Fork Path is quite similar to that of Path ORAM and Fork Path. However, Fork Path focus on the overlapped part between read and write phases when applied on traditional ORAM designs, while the feature of Tiny ORAM requires us to also apply our optimization on the consecutive read operations. With the following rules,

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

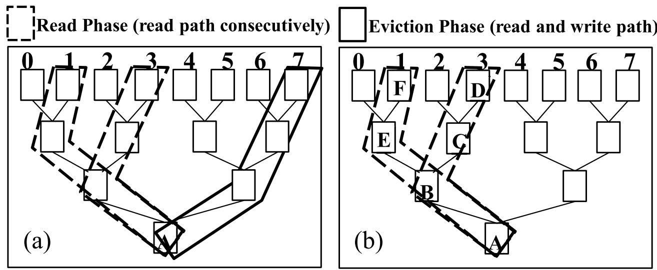


Fig. 10. (a) Access reduction after eviction phase. (b) Access reduction in read phase.

we proved that our design is applicable to the state-of-the-art ORAM designs as well. Rules are summarized as follows.

1) *Rule-1:* The overlapped part of consecutive path reads are loaded only once.
2) *Rule-2:* The overlapped part of consecutive write path and read path are not evicted and loaded, which is similar with that in Path ORAM.
3) *Rule-3:* Dummy label replacing, MAC, and prefetched request insertion are performed similarly.
4) *Rule-4:* Request scheduling should not affect the access pattern of Tiny ORAM as shown in the Fig. 2(b).
5) *Rule-5:* Only read phases are scheduled following the same rule of Path ORAM's to maximize the overlapped part between path accesses. Eviction phases are not affected.

As shown in Fig. 10, without loss of generality, we suppose that A is 2, which means the eviction phase is triggered every after two read phase. The dotted line represents two path reads in the read phase and the solid line denotes the eviction phase (also known as one path read and one path write). In Fig. 10(a), we suppose the path-7 is being written in an eviction phase. Similar to that in Path ORAM, since we know that block A will be written to the ORAM and then loaded by path read of path-1 or path-3, we can omit this kind of accesses without loss of security. In Fig. 10(b), original Tiny ORAM reads block A and block B twice. In this article, the overlapped part of any two path reads is only read once, which can significantly reduce the access overhead.

In summary, accesses to the overlapped part in consecutive path read or between path write and path read are reduced. We have reason to believe that the Fork Path optimization is still effective in the Tiny ORAM scheme, and matching the experimental results are given in Section V as well.

H. Security Proof

The security proof of our Fork Path ORAM relies on a fact that Path ORAM, as the baseline of our design, is proved to be secure. As is mentioned in previous works [18], the security of Path ORAM relies on the independence and randomness of the label sequence, which can hide the original memory access pattern. Through the process of Fork Path ORAM, we leaks no more information about the memory access pattern than traditional Path ORAM design. In path merging, our new modification is only based on the label sequence we access on external memory. Under our assumptions, the keeper of the external memory are not trusted and the access pattern can be easily obtained by the service provider. For the external memory, when you write a block back and then read it immediately to change it, this operation is clearly able to be removed, and of course leaks no information. Path merging removes operations like this in the original label sequence. In other words, we just take advantage of the public information, which is sooner or later revealed to the external memory, to achieve optimized results. As for other optimization methods, including dummy request replacing, ORAM request scheduling, and prefetched request inserting, these are all applied in the local secure processor that is considered to be trusted, which means that the operations are not realized by the external memory. In addition, the leakage of LLC intensity information we mentioned before can also be avoided as long as we keep the scheduling queue full. Moreover, the security of treetop caching is proved in previous work [10]. Our MAC performs quite similar, so the security of our caching scheme can be proved in the same way.

Stash overflow is also an important part of security considerations. In traditional Path ORAM designs, all buckets along the path are loaded into stash and written back in one access. When path merging is applied, the overlapped part of the previous path remains in stash and only the nonoverlapped part of the path is written back and replaced by the incoming access. Hence the memory size in stash in both situations are the same, which means that path merging does not increase the possibility of stash overflow. Similarly, label scheduling will not change the possibility of stash overflow either. The scheduling process reorders the request from LLC, with some dummy requests or prefetched requests inserted. The possibility of stash overflow keeps the same regardless of stash hit or miss [10], since it is only related to the level of the ORAM tree and size of the stash.

The application of Fork Path in Tiny ORAM can also be proved to be secure. We can make similar proof from the above perspectives.

1) The accesses to the overlapped part of write path in eviction phase and read phase can be removed without security loss, which is totally the same with that in Path ORAM.
2) The overlapped part of path reads in read phases can be accessed only once without harming security. This is because the paths to access and how these paths are overlapped are public sooner or later. Hence reducing the accesses to the overlapped part leaks no information.
3) The probability of stash overflow is not affected after deploying Fork Path optimization. As addressed in Rule-4 in Section III-G, eviction phases are not scheduled and still follow a reverse lexicographical order [19], [29]. Thus, the number of blocks to be evicted from the stash to the memory is statistically the same with that of original Tiny ORAM.
4) Scheduling and caching mechanisms are secure, which can be proved similarly to that of Path ORAM.

In conclusion, by applying our optimization techniques, we can prove that in our design, no information about the access pattern is leaked and the possibility of stash overflow

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

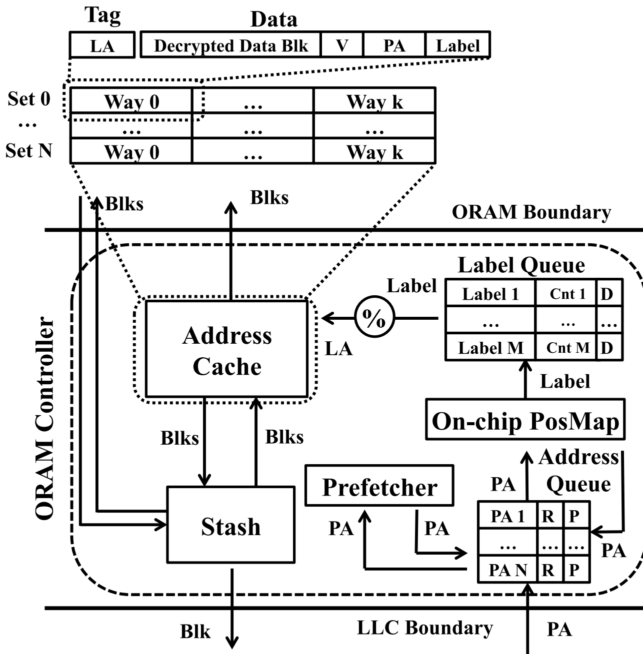


Fig. 11. Architecture of the ORAM controller.

is kept unchanged compared to traditional Path ORAM. Fork Path scheme is secure to be applied, both on traditional Path ORAM, and Tiny ORAM (as a representative of state-of-the-art Path ORAM optimizations).

IV. Fork Path ORAM Architecture

Based on the architecture of traditional ORAM controller [7], [9], we designed a detailed structure of our Fork Path ORAM controller. As shown in Fig. 11, in addition to the stash and position map that already exists in traditional Path ORAM designs, we added two request queues, a set-associative cache and a prefetcher to implement our architecture.

The real memory requests from LLC are forwarded and buffered into the "address queue." Each memory request is stored as a program address (PA), with two extra bits *R* and *P*. Bit *R* is added to identify whether the data in the entry is ready, and bit *P* indicates whether the request is a prefetched request, which may be flushed or replaced later. A prefetcher is added to prefetch memory requests into the address queue according to the rules we mentioned in Section III-E. The requests in the address queue are sent in order to the position map and transformed into an ORAM request sequence. The ORAM requests are sent to another queue named "label queue" where the requests to blocks are stored as their corresponding path labels. An extra bit "D" is inserted to indicate if the request is dummy, which is needed when we need to replace dummy requests with incoming real data requests. Also, each entry of label queue holds a "Cnt" part to memory the "age" of a request. If the Cnt of an ORAM request reaches a threshold, it is very likely to be a real request since dummy ones are supposed to be replaced already. To avoid starvation on data requests that are always at low priority, requests that arrives certain Cnt are promoted to the head of the queue and launched immediately, no matter the request is a dummy one or a real data request.

Processes, including path merging and request scheduling, are performed based on the label queue. As we mentioned, data hazard introduced by request scheduling needs to be considered in our design. By applying proper constraints to the address queue, data hazard can be prevented in the following four possible scenarios.

1) *Read-Before-Read:* If two requests requires the same block in memory, no extra operation is needed since the scheduling process has no effect on the requests.
2) *Read-Before-Write:* If a read request is followed by a write request to the same block, the write block cannot be sent to label queue until the R bit of the read request is set.
3) *Write-Before-Read:* If a write request is followed by a read request to the same block, a data forwarding path is built and the read request is returned directly.
4) *Write-Before-Write:* If a write request is followed by a write request to the same block, the previous write request is flushed.

Note that all the solutions on data hazard is applied in the address queue holding the real data request and the scheduling performs in the label queue. Following the methods above, every request that is sent to the label queue can be scheduled without concerning about data hazard problems, which is also in line with our application of scheduling on the entire label queue.

Data blocks in the merging-aware cache are stored decrypted and can be prompted back to stash when a cache hit happens. 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. For a block at *addr*, we use $level-x$ to denote its level and it is the yth 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 where 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

$$\text{Set_number} = \frac{(2^{x-m_1}-2)*Z}{\text{cache_ways}} + \frac{(y\%2^{x-m_1+1})*Z}{\text{cache_ways}}. \quad (1)$$

The external memory is always stored in DRAM in practical applications. Due to the long access latency per ORAM request, in most of the time the latency introduced by the ORAM controller can be overlapped in our design. In other words, the ORAM controller can function in parallel with the DRAM accesses. With the MAC, data forwarding and our prefetching technique, some requests may even complete without DRAM request returns.

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

TABLE I
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

TABLE II
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

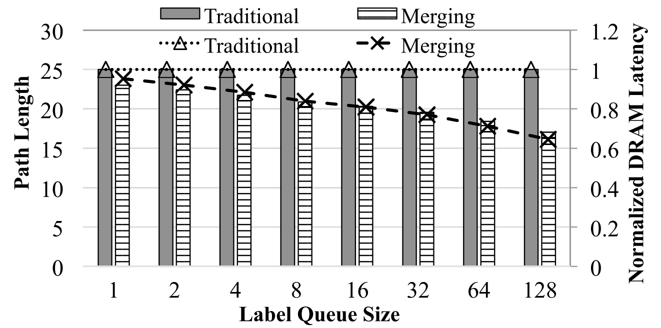


Fig. 12. Average ORAM path length and average DRAM latency (marked as "Δ" and "×") with different label queue sizes.

V. EVALUATION

A. Experimental Setup

We conducted our evaluation in the environment where gem5 [31] is integrated with DRAMSim2, which [32] is used to model the detailed memory accesses of the ORAM tree. We derive the default latency and energy parameters of DDR3 from DRAMSim2. The detailed configuration of processor, ORAM controller, and main memory are summarized in Table I below.

Energy consumptions of ORAM control logic and cache are generated from logic synthesis tool of Synopsys [33] and CACTI [34]. Similar to prior works [7], [9], two memory channels are adopted in the design. In order to maintain a low probability of stash overflow, a 50% memory utilization is presumed [7]. In addition, to maximize the utilization of DRAM bandwidth, a subtree layout [7] is adopted.

Our multiprogrammed workloads are selected from SPEC 2006 [35]. To ensure a comprehensive evaluation, we mix the benchmarks to simulate data access patterns in different scenarios. The benchmarks are divided into a high ORAM overhead group (HG) and a 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. The benchmarks are listed in Table II below.

B. Evaluation With Path ORAM

In this part, we evaluated the performance of Fork Path ORAM compared with traditional Path ORAM designs. The detailed experimental result on the multiprogrammed workloads of a four core configuration are presented below.

1) ORAM Performance Evaluation: Compared with traditional Path ORAM, the application of path-merging leads to the reduction of the average length of ORAM path per memory request, and due to the further optimizations (e.g., request scheduling), the optimization effect is further increased with the expansion of the label queue size (because request scheduling is applied on the entire label queue). Fig. 12 compares the average length of ORAM tree path after applying path merging and request scheduling (labeled as "merging") with the baseline Path ORAM (labeled as "Traditional ORAM") with different label queue sizes. For traditional Path ORAM, the length of ORAM path is fixed, which equals the height of tree in external memory. A total path from leaf to root is always needed to be accessed. When the label queue size is set to 1, which means that only path merging is applied, the expectation of ORAM path length decrease can be proved to be 1. By path merging and request scheduling, the average length of the accessed ORAM path decreases linearly with *log*(*Label Queue size*). The latency of DRAM per request is reduced as well along with the reduction in the number of memory visits. Actually, the reduction of DRAM latency is even more significant than that of the path length, since because the DRAM row-buffer miss rates also decreases with the length of ORAM path.

As we mentioned in Section III-C, lower request latency leads to extra dummy requests, especially when the memory intensity is low. We shown in Fig. 13 the total number of ORAM requests in different label queue sizes, compared with traditional ORAM implementation. Dummy requests are launched to fill the gaps in data request sequence and are used to fill the idle entries during request scheduling. Thus, we can observe that the number of ORAM requests increases with the Label Queue size, significantly with benchmarks in low memory intensity (e.g., over 25% for Mix2). On average, the total number of ORAM requests is increased by only 5% even

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

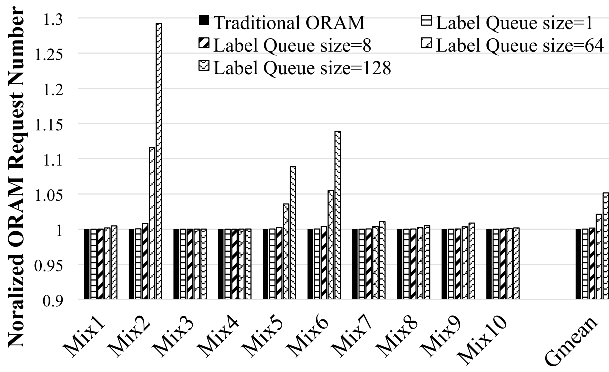


Fig. 13. Normalized total number of ORAM requests.

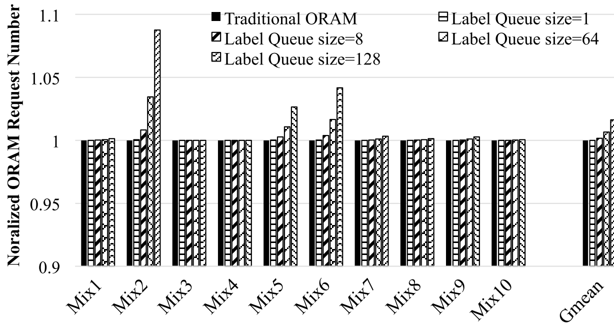


Fig. 14. Normalized total number of ORAM requests after prefetching is applied.

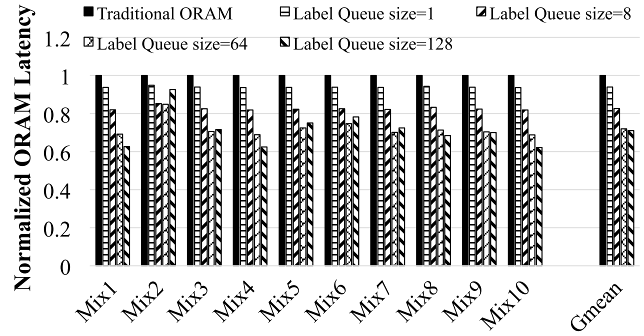


Fig. 15. ORAM latency with different label queue sizes.

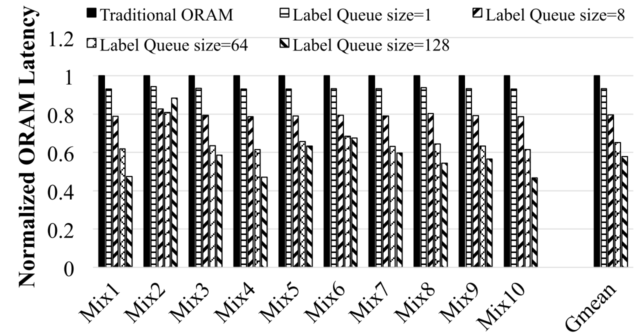


Fig. 16. ORAM latency with different label queue sizes after prefetching is applied.

for a Label Queue size of 128. This is quite acceptable in our design, and the increased ratio is proved that can be further reduced by subsequent optimization.

Fig. 14 illustrates the number of ORAM requests after prefetching is applied. Request prefetching can benefit our design especially when the label queue size is large. When the label queue size is 128, the number of requests can decrease as much as 14% compared to that without prefetching (e.g., Mix2 in Fig. 13). The prefetch operation combined with dummy request replacing can reduce the ratio of dummy requests significantly. On average, the normalized request number drops from 105% to 102% when the label queue size is 128 after prefetching is applied.

In order to provide a comprehensive and straightforward standard of ORAM performance, we introduce a metric called average data request ORAM latency (shorten as ORAM latency) in our evaluation, which represents the completion time of an LLC request since it enters the ORAM controller. ORAM latency can reflect both the reduction in memory traffic and queuing latency, and is supposed to be a feasible standard of ORAM overhead.

In Fig. 15, the ORAM latencies with the application of path merging and request scheduling are presented with increasing label queue sizes on different workloads. It is worth mentioning that as the queue size increases, ORAM latency decreases at first while increased when the queue size is increased from from 64 to 128 on some workloads (which can be offset with request prefetching). In these cases, the benefits of path length reduction has been offset by the extra dummy requests induced. Thus the size of label queue should be carefully designed based on actual storage situation.

The ORAM latencies of ORAM with prefetching is listed in Fig. 16. Compared to Fig. 15, we can find that the prefetching can further enhance the benefit of large queue sizes. Note that the ORAM latency is counted only when the request enters the address queue which activates queued prefetching requests. On average, compared to label queue of 64, when the label queue size is 128, the ORAM latency can decrease by 10%. If we compare Figs. 15 and 16, we can find that the ORAM latency can decrease by as much as 18% on average, when the label queue size is 128. The result is significant not only because of the reduction of dummy requests but also because prefetching itself can shorten the request processing. Thus, it indicates that we can use 128 as default value in the rest of this article.

The efficiency of MAC is evaluated in Fig. 17. Apparently, ORAM latency is reduced after using on-chip caching. Compared to treetop caching, merging-aware caching (labeled as MAC) can further reduce ORAM path length and consequently, achieving a further reduction in ORAM latency. We vary MAC sizes from 128 K bytes to 1 M bytes and compare them to the case using 1 M bytes treetop caching. On average, using MAC can achieve a reduction in ORAM latency comparable to treetop caching with only about 1/4 of cache size.

2) Full System Evaluation: With all optimization techniques used, we make a full system evaluation of our design. The label queue size is set to be 128, as mentioned in Section V-B1. The cache size of MAC is set to 128, KB 256 KB, and 1 MB while the cache size of treetop caching is

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

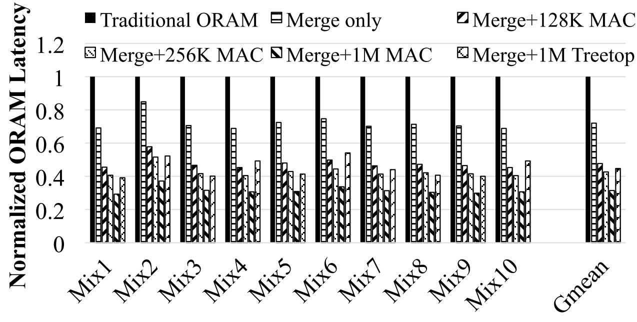


Fig. 17. ORAM latency with different caching designs.

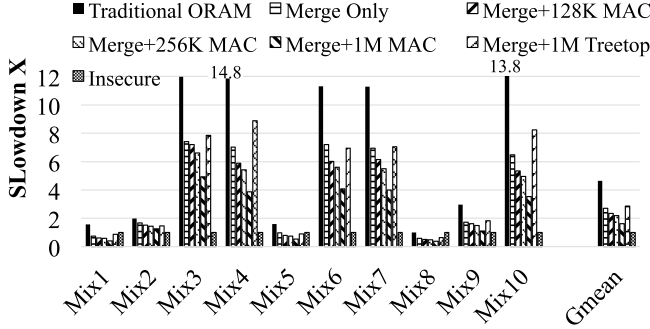


Fig. 18. Slowdown of full system execution time.

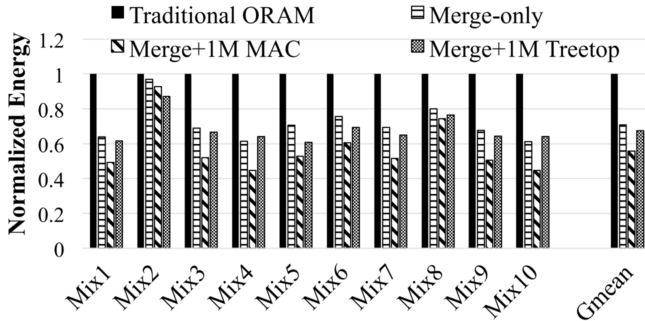


Fig. 19. Energy consumption of ORAM memory system.

fixed at 1 MB. Fig. 18 presents the results of the slowdown of program execution time and also provide that in insecure processor as a reference. As a result, with a 1-MB MAC cache, system execution time reduces by 65% and 43%, compared to the traditional ORAM and that using a 1-MB treetop caching.

Moreover, Fork Path ORAM can also help reducing energy consumption of memory accesses, including both external memory and ORAM controller. As is shown in Fig. 19, the energy consumption is reduced by about 44% 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 17.4% energy reduction.

C. Evaluation With Tiny ORAM

As addressed in Section III-G, our Fork Path optimizations can also be applied to Tiny ORAM [19] or other MRSE ORAMs such as Ring ORAM [29]. Due to the page limit,

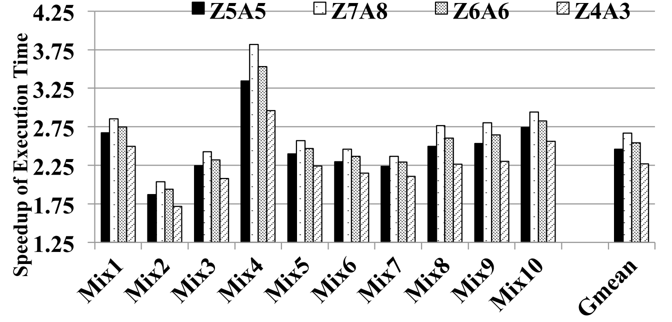


Fig. 20. Speedup with different Z, A settings of Tiny ORAM.

we only illustrate the results of performance speedup with our optimizations, which we think is the most important result to demonstrate efficiency. The other results are similar to those of Path ORAM. For simplicity, we set the label queue size as 128 and MAC size as 1 MB as in the previous section.

Fig. 20 shows the speedup of optimized Tiny ORAM with different bucket sizes (Z) and eviction rates (A) compared to the original Tiny ORAM. We choose these settings since they are proposed as the most promising settings and comprehensively evaluated/discussed in [19]. We can find that with larger bucket size or a larger eviction rate, the speedup is higher. This is mainly because: 1) larger bucket size can reduce the path length, leading to a higher ratio of $(path_{merged}/path_{total})$. Thus, the ORAM latency can be better reduced and 2) larger eviction rate results in a higher percentage of overlapped path during read phase. On average, Fork path scheme can accelerate the Tiny ORAM up to 2.7× when $(Z, A) = (7, 8)$.

VI. Related Work

ORAM algorithms are first proposed by Goldreich and Ostrovsky [4], [5] around 30 years ago. Since its proposal, efforts on finding a practical ORAM scheme have been made due to the large overhead introduced by ORAM [6], [8], [16]–[18]. Recently, Path ORAM has attracted attentions from researchers because of its simplicity in algorithms and efficiency in reducing memory access overhead. As we stated below, several follow-up techniques have been proposed these years based on Path ORAM.

Ren *et al.* [7] proposed several optimization techniques for basic Path ORAM, including background eviction, static super block, and subtree layout. Maas *et al.* demonstrated Phantom [10]—the first hardware implementation of Path ORAM, in which treetop caching and min-heap eviction are proposed to reduce the latency of path accesses and stash operations. Fletcher *et al.* [30] proposed a dynamic scheme to protect the timing channel of ORAM accesses. Freecursive ORAM [9] is presented by the same group later where PosMap Lookaside Buffer (PLB) and PosMap compression are introduced to mitigate the overhead of PosMap accesses. Yu *et al.* proposed PrORAM [11] in which dynamic prefetching is introduced. Compared to static prefetching, dynamic prefetching is more flexible to join or disjoin adjacent blocks according to the program's locality.

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

Except in ORAM protocol and ORAM controller, new security assumptions or new technologies are applied to further optimize ORAM. Wang *et al.* [20] proposed CP-ORAM which schedules secure requests and insecure requests to improve the server performance. Shafiee *et al.* [22] further mitigated the overhead of Path ORAM with architectural optimizations of DRAM, including bucket splitting and parallelized DRAM accesses which are based on a secure buffer. Aga and Narayanasamy [21] introduced a 3-D-stacked new structure of memories to further improve ORAM performance, which enables DRAM capable of secure computation. These optimizations are orthogonal and can be directly applied to Fork Path ORAM.

VII. Conclusion

Due to the security requirements of cloud computing, ORAM has been applied extensively in secure processors. However, the overhead of memory operations is hard to ignore and has become the bottleneck of its application. According to our observation, a large amount of redundant memory accesses still exist even in the most practical known-to-date ORAM scheme, which can be removed without harming the security ORAM provides. We propose path merging and request scheduling to remove the redundant operations. Based on these two optimization methods, we further propose dummy label replacing, request prefetching, and MAC to improve efficiency. Experiments with Path ORAM and Tiny ORAM show that Fork Path ORAM brings a significant performance enhancement and can be flexibly applied in various Path ORAM designs.

References

[1] X. Zhuang, T. Zhang, and S. Pande, "Hide: An infrastructure for efficiently protecting information leakage on the address bus," in *Proc. ACM SIGPLAN Notices*, vol. 39, no. 11, 2004, pp. 72–84.

[2] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "GhostRider: A hardware–software system for memory trace oblivious computation," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2015, pp. 87–101.

[3] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *Proc. 24th USENIX Security Symp. (USENIX Security)*, Aug. 2015, pp. 431–446. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[4] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *Proc. 19th Annu. ACM Symp. Theory Comput.*, 1987, pp. 182–194.

[5] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[6] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," in *Proc. 23rd Annu. ACM SIAM Symp. Discr. Algorithms*, 2012, pp. 157–167.

[7] 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," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 571–582, 2013.

[8] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," *CoRR*, vol. abs/1106.3652, 2011.

[9] 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 *Proc. 12th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2015, pp. 103–116.

[10] M. Maas *et al.*, "PHANTOM: Practical oblivious computation in a secure processor," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2013, pp. 311–324.

[11] X. Yu *et al.*, "PrORAM: Dynamic prefetcher for oblivious RAM," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, Portland, OR, USA, 2015, pp. 616–628.

[12] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, San Diego, CA, USA, 2003, p. 351.

[13] W. Shi and H.-H. S. Lee, "Authentication control point and its implications for secure processor design," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchit.*, Orlando, FL, USA, 2006, pp. 103–112.

[14] G. E. Suh, D. Clarke, B. Gasend, M. V. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, San Diego, CA, USA, 2003, p. 339.

[15] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proc. IEEE 37th Int. Symp. Microarchit. (MICRO-37)*, Portland, OR, USA, 2004, pp. 221–232.

[16] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log n)^3)$ worst-case cost," in *Advances in Cryptology–ASIACRYPT 2011*. Heidelberg, Germany: Springer, 2011, pp. 197–214.

[17] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious RAM and a new balancing scheme," in *Proc. 23rd Annu. ACM SIAM Symp. Discr. Algorithms*, Kyoto, Japan, 2012, pp. 143–156.

[18] E. Stefanov *et al.*, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, Berlin, Germany, 2013, pp. 299–310.

[19] C. W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, and S. Devadas, "RAW path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification," Rep., 2014.

[20] R. Wang, Y. Zhang, and J. Yang, "Cooperative path-ORAM for effective memory bandwidth sharing in server settings," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Austin, TX, USA, 2017, pp. 325–336.

[21] S. Aga and S. Narayanasamy, "InvisiMem: Smart memory defenses for memory bus side channel," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Toronto, ON, Canada, 2017, pp. 94–106.

[22] A. Shafiee, R. Balasubramonian, M. Tiwari, and F. Li, "Secure DIMM: Moving ORAM primitives closer to memory," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Vienna, Austria, 2018, pp. 428–440.

[23] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *Proc. IEEE Comput. Archit. Int. Symp.*, Madison, WI, USA, 2005, pp. 14–24.

[24] D. Lie *et al.*, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[25] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. 17th Annu. Int. Conf. Supercomput.*, San Francisco, CA, USA, 2003, pp. 160–171.

[26] 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 *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, Chicago, IL, USA, 2007, pp. 183–196.

[27] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proc. IEEE 9th Int. Symp. High Perform. Comput. Archit. (HPCA-9)*, Anaheim, CA, USA, 2003, pp. 295–306.

[28] C. W. Fletcher, M. V. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proc. 7th ACM workshop Scalable Trusted Comput.*, Raleigh, NC, USA, 2012, pp. 3–8.

[29] L. Ren *et al.*, "Ring ORAM: Closing the gap between small and large client storage oblivious RAM," *IACR Cryptol. ePrint Archive*, vol. 2014, p. 997, 2014.

[30] 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 *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Orlando, FL, USA, 2014, pp. 213–224.

[31] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.

[32] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan.–Jun. 2011.

[33] H. Bhatnagar, *Advanced ASIC Chip Synthesis: Using Synopsys Design Compiler™ Physical Compiler™ and PrimeTime* . New York, NY, USA: Springer, 2007.

[34] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model," Compaq Comput. Corporat., Palo Alto, CA, USA, Rep. 2001/2, 2001.

[35] J. L. Henning, "Spec CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

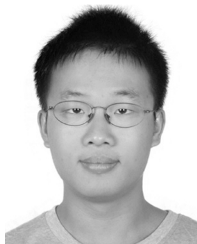
Jingchen Zhu (S'02) is currently pursuing the undergraduation degree with Peking University, Beijing, China.

He was with the Center for Energy-Efficient Computing and Applications, Peking University. His current research interests include accelerator design and trusted computing.

Guangyu Sun (M'10) received the B.S. and M.S. degrees from Tsinghua University, Beijing, China, in 2003 and 2006, respectively, and the Ph.D. degree in computer science from Pennsylvania State University, State College, PA, USA, in 2011.

He is currently an Assistant Professor of CECA with Peking University, Beijing. His current research interests include computer architecture, VLSI Design, and electronic design automation. He has published over 60 journals and refereed conference papers in the above areas.

Dr. Sun has also served as a peer reviewer and technical referee for several journals, which include the IEEE MICRO, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, and the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. He is a member of CCF.



Xian Zhang (S'07) received the B.S. degree from Peking University, Beijing, China, in 2013, where he is currently pursuing the Ph.D. degree with the Center for Energy-Efficient Computing and Applications.

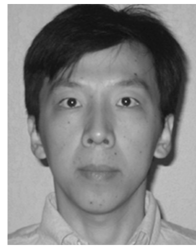
His current research interests include blockchain, cryptography, and trusted computing.

Chao Zhang (S'09) received the B.S. degree in microelectronics and the Ph.D. degree in computer science from Peking University, Beijing, China, in 2012 and 2017, respectively.

He is currently with Amazon, Beijing. His current research interests include architectural optimization for STT-RAM and domain wall racetrack memory.

Weiqi Zhang (S'07) received the B.S. degree in CS and the M.S. degree in computer architecture from Peking University, Beijing, China, in 2014 and 2017, respectively.

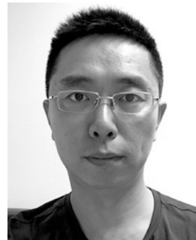
He is currently a Technology Teacher with Beijing National Day School, Beijing. His current research interests include storage systems and nonvolatile memories.



Yun Liang (M'10) received the B.S. degree in software engineering from Tongji University, Shanghai, China, in 2004, and the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2010.

He was a Research Scientist with the University of Illinois Urbana–Champaign, Champaign, IL, USA, from 2010 to 2012. He has been an Assistant Professor with the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, since 2012. His current research interests include heterogeneous computing, embedded system, and high level synthesis.

Dr. Liang was a recipient of the Best Paper Award in International Symposium on Field-Programmable Custom Computing Machines in 2011 and the Best Paper Award Nominations in CODES+ISSS'08, FPT'11, DAC'12, and ASPDAC'16. He serves as a Technical Committee Member for Asia South Pacific Design Automation Conference, Design Automation and Test in Europe, International Conference on Compilers Architecture and Synthesis for Embedded System, International Conference on Computer Aided Design, and International Conference on Parallel Architectures and Compilation Techniques.



Tao Wang (M'10) received the B.S. and Ph.D. degrees from Peking University, Beijing, China, in 1999, and 2006, respectively.

He is currently an Associate Professor with Peking University. His current research interests include computer architecture, reconfigurable logic, wireless network architecture, and mobile cloud computing.

Yiran Chen (F'10) received the B.S and M.S. degrees (Hons.) from Tsinghua University, Beijing, China, and the Ph.D. degree from Purdue University, Beijing, in 2005.

He joined the University of Pittsburgh, Pittsburgh, PA, USA, as an Assistant Professor, in 2010 and then promoted to a Associate Professor with tenure in 2014, held Bicentennial Alumni Faculty Fellow. He is currently an Tenured Associate Professor with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA, where he serves as the Co-Director of Duke Center for Evolutionary Intelligence.

Dr. Chen was a recipient of the 5 Best Paper Awards and 15 best paper nominations from international conferences, the NSF CAREER Award and the ACM SIGDA Outstanding New Faculty Award. He is the Associate Editor of the IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the IEEE DESIGN AND TEST OF COMPUTERS, IEEE EMBEDDED SYSTEMS LETTERS, the *ACM Journal of Emerging Technologies in Computing Systems*, the *ACM Transactions on Cyber-Physical Systems*, and served on the technical and organization committees of over 40 international conferences.

Jia Di (SM'10) received the B.S. and M.S. degrees in electrical engineering from Tsinghua University, Beijing, China, in 1997 and 2000, respectively, and the Ph.D. degree in electrical engineering from the University of Central Florida, Orlando, FL, USA, in 2004.

In Fall 2004, he joined the Computer Science and Computer Engineering Department, University of Arkansas, Fayetteville, AR, USA, where he is currently a Professor and 21st Century Research Leadership Chair. He has published 1 book and over 100 papers in technical journals and conference proceedings. He also holds five U.S. patents. His current research interests include asynchronous integrated circuit design and hardware security.

Prof. Di is an Elected Member of the National Academy of Inventors.

Authorized licensed use limited to: Peking University. Downloaded on April 07,2021 at 01:17:40 UTC from IEEE Xplore. Restrictions apply.