# H-ORAM: A Cacheable ORAM Interface for Efficient I/O Accesses

Liang Liu†, Rujia Wang‡, Youtao Zhang†, Jun Yang†
† University of Pittsburgh, ‡Illinois Institute of Technology
{lil125,youtao,juy9}@pitt.edu,rwang67@iit.edu

## ABSTRACT

Oblivious RAM (ORAM) is an effective security primitive to prevent access pattern leakage. By adding redundant memory accesses, ORAM prevents attackers from revealing the patterns in the access sequences. However, ORAM tends to introduce a huge degradation on the performance. With growing address space to be protected, ORAM has to store the majority of data in the lower level storage, which further degrades the system performance.

In this paper, we propose Hybrid ORAM (H-ORAM), a novel ORAM primitive to address large performance degradation when overflowing the user data to storage. H-ORAM consists of a batch scheduling scheme for enhancing the memory bandwidth usage, and a novel ORAM interface that returns data without waiting for the I/O access each time. We evaluate H-ORAM on a real machine implementation. The experimental results show that that H-ORAM outperforms the state-of-the-art Path ORAM by 20×.

## KEYWORDS

Oblivious RAM, memory security, I/O accesses, scheduling, oblivious shuffle

**ACM Reference Format:**
Liang Liu†, Rujia Wang‡, Youtao Zhang†, Jun Yang†. 2019. H-ORAM: A Cacheable ORAM Interface for Efficient I/O Accesses. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317841

## 1 INTRODUCTION

Modern trusted hardware, e.g., TPM [3], SGX [7], and XOM [8], secure the processing of sensitive data through data encryption and integrity check, which effectively prevent adversaries from revealing the plaintext or compromising the data.

However, information may be leaked through various side channels during execution. For instance, the timing information[18], memory access patterns[6] and the power usage[1] are also the accessible sources for the malicious adversaries. By observing or tampering with the sources above, attackers can retrieve sensitive data without directly reading the data contents. For example, researchers have discovered that on a remote data storage server with searchable encryption, access pattern can still leak a significant amount of sensitive information using a little of prior knowledge[6].

Oblivious RAM (ORAM) is a security method primitive initially proposed by Goldreich and Ostrovsky to hide the memory access pattern entirely from adversaries[4]. Several ORAM protocols have been developed since then. They share the same design philosophy: multiple dummy accesses or dummy data blocks need to be padded with actual data access, and the address needs to be reshuffled periodically to achieve random accesses. The adversaries can only observe a list of memory addresses being accessed, but they cannot correctly guess where the actual sensitive data is. Moreover, since the actual access pattern should be concealed to the malicious adversaries but revealed to the controller, the designer needs a protected area in the hardware to store the essential sensitive information. In the section 2.2, we will discuss the details of the most representative ORAM schemes.

According to the major invariants of ORAM, to achieve obliviousness, a single data access will couple with tens or hundreds of dummy requests. When the ORAM size is small, the entire ORAM protected data can be entirely loaded into the memory. However, with the increase of data set size, the ORAM capacity will also increase linearly. In such case, the main memory is no longer capable of storing such a large amount of data. Recently, several researchers[2, 13] propose to extent ORAM to the storage level to achieve access pattern protection with larger capacity offered, such that every data request needs to obliviously access both memory and storage, which adds the I/O access overhead to the original ORAM access overhead.

In this paper, we present H-ORAM, a novel hybrid ORAM scheme targets at the slow I/O access bottleneck during ORAM accesses when the ORAM dataset is split in the memory and storage. This work tends to accomplish the following goals while still ensure the security, 1) Construct an ORAM interface with the cache function enabled: the cache is capable of improving the ORAM access time by removing unnecessary I/O accesses, without leaking access pattern by lightweight eviction and shuffle. 2) Decrease the data storage overhead. Our ORAM construction is more compact and uses less space compared to other ORAM protocols. We describe the ORAM background and basics in Section 2, elaborate our motivation in Section 3, describe the details of our proposed H-ORAM in Section 4. Our theoretical and experimental results are shown in Section 5, and we conclude this paper in Section 6.

## 2 BACKGROUND

### 2.1 Threat Model

Our threat model is similar to most of threat models that need ORAM to protect[11, 12, 14, 17]. We assume that the victim application is safe and the attacker can observe the access pattern of


*DAC '19, June 2–6, 2019, Las Vegas, NV, USA*

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317841

the victim application. We do not consider the side channel information leakage across multiple applications/ users, and the ORAM dataset is private. The application could be run on a computing node with secure hardware such as SGX so that the processor, as well as the on-chip cache, is protected and trusted. Although SGX suffers from side channel attacks, mitigate the vulnerabilities is orthogonal to our work. Therefore, in our model, we focus on the off chip memory and I/O traffic. With encryption, we can preserve data privacy, however, the access pattern may still be observed if the attacker is able to tamper the memory bus. Another scenario is that the application is running on the secure local machine, and it is accessing a remote storage server. The user outsourced data to the cloud storage vendor and the communication (load and store) patterns could leak information.

## 2.2 ORAM Basics

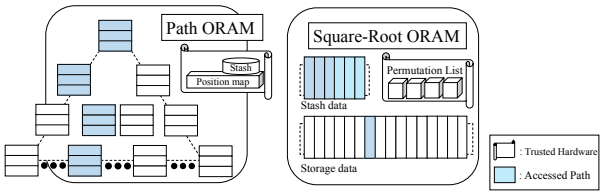


**Figure 1: Basic ORAM schemes.**

Oblivious RAM protects the system from access pattern leakage by randomly remapping the address after access. Path ORAM [15] is one of the most simple and practical ORAM protocol, which organizes the memory as a tree-like layout, as shown in Figure 1 left. Encrypted data can be stored at each node of the tree, and the ORAM controller translates each access into a path access with $O(\log N)$ access overhead. After fetching the data inside of the secure ORAM controller, the accessed data will be decrypted and remapped to a different path. Therefore, repeatedly accessing the same data will not reveal the same access pattern on the memory bus. Stash, position map, as well as other components in ORAM controller, need to be stored in securely. Path ORAM requires extra storage space to store dummy blocks, and the best utilization rate is around 50%[15], so storing $N$ real blocks requires $2N$ space.

A different type of ORAM organizes the memory space as a flat space. Square-root ORAM [5, 19] maintains a permutation list which stores the mapping between the physical address and virtual address, as shown in Figure 1 on the right. To initiate, $N$ data blocks need to be padded with $\sqrt{N}$ dummy blocks and reshuffled, to generate the permutation list. When a block is not found in the stash, the ORAM controller fetches the data from memory, and when the data is found in the stash, a dummy block also need to be fetched from the memory to avoid information leakage. After $T$ accesses, all dummy blocks in the stash and need to be removed and the whole structure need to be shuffled and re-initialized. Compared with Path ORAM with $O(\log N)$ access overhead, square root ORAM requires a $O(\sqrt{N})$ of access overhead. Partition ORAM [14] also uses flat memory organization and divides the storage into multiple partitions to reduce the shuffle overhead each time with more frequent shuffle operations.

# 3 MOTIVATION

## 3.1 Limitations of current ORAM designs

Except from the access overhead brought by ORAM protocols, current ORAM designs do not consider the deep memory and storage hierarchy in modern computing systems. For example, in ZeroTrace[13], when the data set is larger than the main memory capacity, they directly extend the leaf nodes to the storage(HDD) backend, as shown in Figure 2 a). The tree-top cache is a straightforward design since most levels are in the fast memory region. However, each path access is translated into multiple fast memory accesses and multiple slow I/O accesses. Also, the tree type organization is hard to adopt other cache techniques because of the low locality. Considering the performance gap between the memory and I/O access, plus the imbalance of memory and I/O usage, such design is inefficient regarding I/O bandwidth overhead.

Using square-root ORAM or partition ORAM in such case will reduce the I/O overhead because each time only one data block needs to be fetched from the storage backend, as shown in Figure 2 b). In addition, the flat memory organization allows efficient caching on the top layer. However, the shuffle operation needs to be performed frequently, and the entire storage needs to wait for the shuffle completed before next ORAM operation.

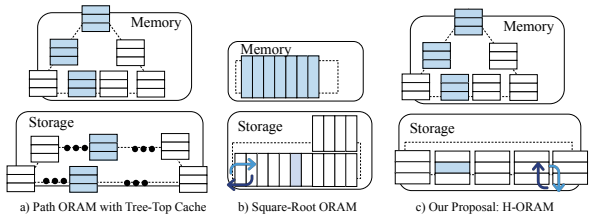


**Figure 2: Derivation from Square-Root ORAM.**

## 3.2 Our design goal

The limitations of the existing ORAM protocols motivate us to design a cacheable ORAM interface with low shuffle overhead.

Considering the basic structure of square-root ORAM, as is mentioned in Section 2.2, the data is chunk into two parts: stash data and storage data. All data in the stash needs to be accessed when there is an ORAM access, which is an $O(\sqrt{N})$ overhead. Accessing the data in the storage only requires a single access, but it needs periodically shuffle. The stash data can be stored in the fast memory. However, the speed of memory is hard to afford $O(\sqrt{N})$ of redundant accesses. Our first design goal is to minimize the in-memory access overhead while remain the obliviousness. We adopt the Path ORAM for the in-memory data storage, and reduce the overhead from $O(\sqrt{N})$ to $O(\log \sqrt{N})$.

Another inevitable overhead of the square root ORAM is the shuffle process. Unlike the naive shuffle algorithm, ORAM requires an oblivious version of shuffle algorithm such as permutation network, cache Shuffle or Melbourne shuffle [9, 10], all of which bring excessive overhead. Our second design goal is to minimize the shuffle overhead by introducing a new lightweight shuffle process delicately designed for ORAM. As a result, with the above approaches, our H-ORAM, can theoretically and experimentally outperform the state-of-the-art Path ORAM design in terms of performance

and storage overhead. The basic sketch of the H-ORAM memory organization is shown in Figure 2c).

# 4 H-ORAM: DESIGN AND IMPLEMENTATION

## 4.1 Design Layout and Data Flow

H-ORAM distributes the data to three different physical layers: a control layer, a memory layer, and a storage layer. The first layer is the secure shelter, which utilizes secure hardware such as Intel SGX, and the operations and data inside of the secure shelter are considered tamper-resistant. The second layer in the middle is in memory and it stores data that can be accessed at a high speed. The third layer stores data in the slow but large storage. The design layout of the three layers is shown in Figure 3.

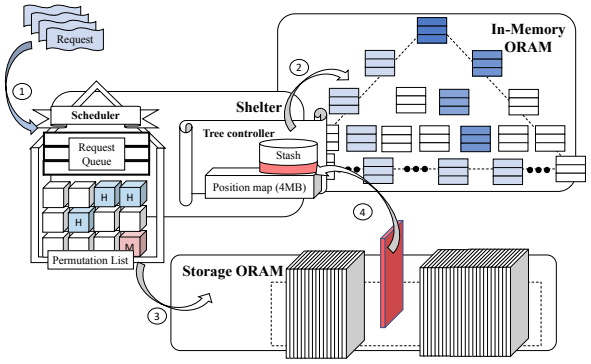


**Figure 3: Design Layout**

**Control layer:** The control layer should be protected by the secure hardware. In H-ORAM, the position map for in-memory Path ORAM and the permutation list for storage side ORAM need to be protected. In addition, the control layer contains the scheduler for secure scheduling.

**Memory layer:** The data inside is organized as a Path ORAM tree, which can store up to $n$ data blocks (up to 50% are real blocks). The in-memory ORAM works as the cache during the H-ORAM access. In the beginning, the tree is empty, and data is brought from the storage to the tree. After $n/2$ blocks have been loaded, the tree is evicted back to the storage and will be reconstructed again.

**Storage layer:** The data inside is organized into $N$ data blocks, each of which stores a small, encrypted and permuted data block. To ensure the security, we need to shuffle all the blocks in the storage periodically.

To serve an ORAM request, the scheduler needs to pick and group requests inside of the Request Queue. The H-ORAM periodically swaps between two periods: access period and shuffle period. Since the in-memory ORAM can support up to $n/2$ I/O accesses before next shuffle, we allow $n/2$ I/O accesses for each access period. For each access, the scheduler will scan the Request Queue from the beginning and fetch $\bar{c}$ requests in the table. (see Section 4.2). Then, the scheduler will firstly check the permutation list for each request. The permutation list records: 1) a Boolean bit represents whether a block is loaded into memory already, 2) its file address if in storage (or the position map id if in memory). After scanning the Request Queue, the scheduler computes 1 I/O address and $\bar{c}$ in-memory path addresses. The I/O fetches the miss data from the storage to the stash of in-memory path ORAM and assign an random leaf id to it. When the group of accesses is finished, the tree access brings the data back to the Request Queue, while the I/O access brings data to the stash of the in-memory path ORAM.

When reaching the $n/2$ limit, the H-ORAM will call the shuffle function. The shuffle period includes three procedures: 1) Evict the path ORAM tree. 2) Shuffle the entire storage data. 3) Initialize a new Path ORAM tree. The detail is shown in the section 4.3.

## 4.2 Secure Scheduler for Cache Purpose

A high hit rate cache can greatly improve the system performance. When it comes to a cache designed for ORAM, it not only needs to provide high performance, but also preserves the system security. In this section, we introduce the scheduler's group strategy and the I/O pre-fetching to improve the hit rate while remaining the oblivious access pattern.

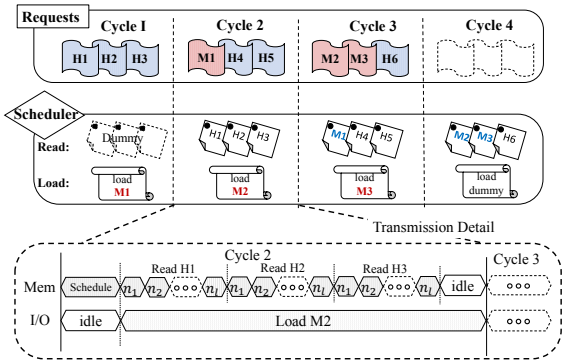


**Figure 4: An Example of Request Scheduler with Pre-fetch**

For the security consideration, the hit or miss information cannot be leaked to the adversaries. In our ORAM design, we use the group strategy to hide the information. The scheduler groups every $\bar{c}$ of the in-coming requests $\{R_1, R_2, \ldots R_{\bar{c}}\}$ as a group. The $\bar{c}$ value depends on the hit-rate of the current stage, and our goal is to make that on average in every group, there are $\bar{c}$ of hit requests and 1 miss. Our scheduler will firstly schedule the I/O load for the miss request, and after the miss request is fetch to memory, we conduct $\bar{c}$ of in-memory reads for the next cycle.

Since there exist variances among the requests, we can not exactly find $\bar{c}$ hit and 1 miss in every cycle, so we have to pad the dummy reads or loads to fill the blank. We further propose an I/O pre-fetching optimization to reduce the dummy requests padded per cycle by early searching next available requests in the Request Queue.

we define a distance $d : d > \bar{c}$ such that during each I/O cycle, the scheduler will scan the next $d$ requests to find a proper match for the current schedule group. Figure 4 shows an example of our group strategy with I/O pre-fetching optimization. In the example, we assume that $\bar{c} = 3$ and $d = 9$. At the first cycle, the scheduler scans all 9 requests in the queue and schedules the first miss request $M1$ as a I/O load task. The missed data will then be loaded into the stash and has its path position recorded in the position map. In the second cycle, three hit in-memory Path ORAM read requests $H1, H2, H3$ and the next miss $M2$ are serviced. After the Cycle 2 is done, the $M1$ is potentially written back to the in-memory ORAM

tree, or still in the stash. Therefore, at Cycle 3, if the $M1$ is in the ORAM, we read the data as a hit request, and if $M1$ is in the stash, we read the corresponding path and write $M1$ back to the in-memory ORAM. The scheduler repeats the same strategy to reorder and group requests, to minimize the number of dummy requests needed per cycle.

Although the hit rate varies across the execution, we use $\overline{c}$ to represent the average hit rate over a constant period. The whole access period can be divided into $s$ stages, and we use different $\overline{c}$ value for different stage to evaluate. At the beginning, the in-memory ORAM is empty, so the $\overline{c}$ is set at a small value. When the in-memory ORAM caches more data, $\overline{c}$ can be set at a bigger value to issue more in-memory hit requests.

## 4.3 Evict and Shuffle

In this section, we discuss how data is managed during the shuffle period of H-ORAM.

**Oblivious Tree Evict:** Since the in-memory data tree is exposed to adversaries, when we call the eviction function, we should ensure its obliviousness (without leaking which block is dummy). Here we design a simple approach: 1) Read all the block (both real and dummy) from tree into a temporary buffer. 2) Run the oblivious shuffle on this buffer. 3) Scan the shuffled buffer and remove the dummy. The reorganized evicted data is shown in read in Figure 5.

**Group and partition shuffle:** The original square-root ORAM has an oblivious shuffle stage which brings too much overhead ($O(4N)$ of I/O overhead)[19]. To reduce the shuffle overhead, we divide the storage into multiple partitions.

Similar to the Partition ORAM[14], we divide the whole data set into $\sqrt{N}$ of partitions and each partition stores $\sqrt{N}$ blocks of data. As is shown in Figure 5, during the shuffle period, the partitions are shuffled sequentially from the left to right. During the $i$th shuffle, the controller firstly reads $i$th partition (cold data) to the memory and concatenates it with the $i$th pieces of evicted data(hot data) and shuffle them as whole. The in-memory shuffle algorithm is free to choose because memory is fast enough, and we use the cache shuffle here. Finally, we write the shuffled partition to the storage and then process the $i+1$th shuffle.

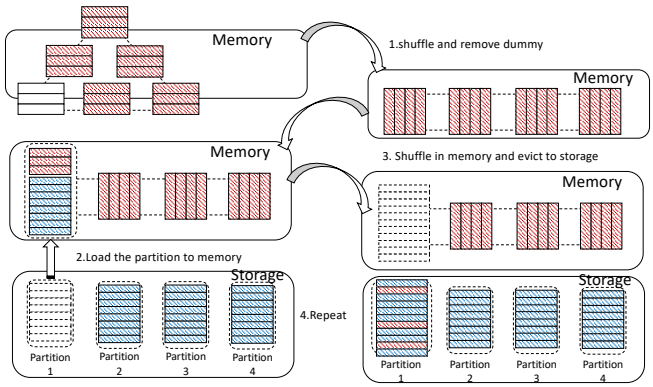


**Figure 5: Evict and Shuffle Stage of H-ORAM**

**Security proof:** The different between the proposed group partition shuffle and the partition ORAM is the order of partition to shuffle. We conduct the shuffle from the first partition to the $\sqrt{N}$ partition :$\{1, 2, \ldots, \sqrt{N}\}$. The partition ORAM conduct the shuffle by randomly choosing a partition $p$, which products a sequence $\{p_1, p_2, \ldots p_{\sqrt{N}}\}$. However, since both of our H-ORAM and partition ORAM provide the unbiased partition access, which ensure the equivalent possibility of access every partition, which is $\forall i \subset \{1, 2, \ldots \sqrt{N}\}, \mathcal{P}(i) = \frac{1}{\sqrt{N}}$. Therefore, the expect value of $i$ th partition to be shuffle for both schemes is equal. Therefore, our proposed partition shuffle has the equivalent security to the partition ORAM shuffle stage.

## 4.4 Security Analysis

Our proposed H-ORAM is secure in the following aspects:

**Access Security:** The H-ORAM achieves highest security protection when conducting in-memory and I/O access. The in-memory access is protected by path ORAM protocol, which is proof secure by randomly changing the location of data. The data fetch from I/O is shuffled after $n$ accesses, and only accessed once per access period, which is also proof secure by the square-root ORAM.

**Scheduler Security:** We use the group strategy to hide the hit or miss information to all parties except the user. The adversaries are not able to infer anything from the hit/miss observed on the memory bus, because each scheduling group has the same hit and miss pattern.

**Shuffle Obliviousness:** Our eviction and shuffle ensures the data is periodically permuted in storage. Our design follows the setting of partition ORAM, and group the evicted data with cold data in storage into $\sqrt{N}$ for partition shuffle, which achieves the equivalent security of partition ORAM.

# 5 RESULTS

## 5.1 Theoretical Analysis

Our baseline is the tree-top-cache path ORAM. In the rest calculation, we denote $N$ as the total amount of block, $Z$ as the bucket size, $n/2$ as the amount of real block in memory. In section 4.2, we define $\overline{c}$ as the number of memory requests serviced when waiting for one I/O request. To simplify the process, we compute the average value $\hat{c}$, which considers the different execution stages, where $c_i n_i$ are the number of the memory requests serviced per stage.

$$\hat{c} = \frac{2}{n}(c_1 n_1 + c_2 n_2 + \ldots c_s n_s)$$

Then, we calculate the I/O overhead of the Path ORAM. Since Path ORAM needs to include dummy data no less than the real data, the total size of baseline Path ORAM to store $N$ real blocks is $2N$. Therefore, the path level is calculated as:

$$\begin{aligned}\text{path level} &= \log_2 \frac{n}{Z} + (\log_2 \frac{2N}{Z} - \log_2 \frac{n}{Z}) \\ &= \log_2 \frac{n}{Z}(\textbf{MEM}) + \log_2 \frac{2N}{n}(\textbf{I/O})\end{aligned}$$

Here, we extract the right most part to calculate the I/O overhead. For the load and store operation, The average I/O overhead of Path ORAM is:

$$(Z \log_2 \frac{2N}{n})\ \textbf{reads} + (Z \log_2 \frac{2N}{n})\ \textbf{writes}$$

In comparison, our H-ORAM fetches 1 block each time during the access period. When finishing $n\hat{c}/2$ I/O accesses, we need to shuffle the entire dataset. During the shuffle period, H-ORAM fetches $N-n$ blocks of data and rewrite $N$ blocks back to storage. Therefore, the average access overhead is:

$$\{1+\frac{2(N-n)}{n\hat{c}}\}\ \textbf{reads}\ +\frac{2N}{n\hat{c}}\ \textbf{writes}$$

By substitute the practical values, we use a moderate Path ORAM parameter where $Z=4$. The result shows that when the ratio $(N/n)$ is small, the H-ORAM can achieve better performance over Path ORAM. For example, when $\hat{c}=4$, and $\frac{N}{n}=8$, we can achieve around $8x$ I/O access overhead reduction.

Table 1 shows a concrete example that we have a 1GB real data set, and the memory can store up to 128 MB ORAM tree. Follow the previous calculation, for each access period, we can conduct 262, 144 I/O requests without shuffle.

$$\frac{n\hat{c}}{2}=\frac{128\times1024\times4}{2}=65536\times4=262,144$$

After the 262, 144 requests finish, the entire dataset is shuffled (see section 4.3), which brings 1 GB (write) + (1GB − 128 MB)(read) I/O accesses. We calculate the average access overhead as follow:

$$\begin{aligned}\text{average access overhead} &= \text{access overhead}+\frac{\text{shuffle overhead}}{\text{number of request}}\\ &= 1\text{KB reads}+\frac{0.875\text{GB (reads)}+1\text{ GB (writes)}}{262144}\\ &= 4.5\text{KB reads} + 4\text{KB writes}\end{aligned}$$

**Table 1: Overhead comparison for one period ( 1 GB data size, 128 MB memory size, 1 KB block size )**

| | H-ORAM | Path ORAM |
|---|---|---|
| **Storage/Memory Size** | 1GB/128MB | 1.875GB/128MB |
| **Path ORAM Level** | 16 | 16 + 4 |
| **Requests Serviced** | 262144 | 65536 |
| **Access Overhead** | 1 KB (read) | 16KB(read) +16KB(write) |
| **Shuffle Overhead** | 0.875 GB(read) + 1 GB(write) | N/A |
| **Overall Overhead** | 1 GB (read) + 1 GB(write) | 4 GB(read) + 4 GB(write) |
| **Average Overhead** | 4.5 KB (read) + 4KB(write) | 16KB(read) + 16KB(write) |

**Discussion on shuffle overhead:** From the above computation, we find that the biggest overhead of our H-ORAM is the shuffle process, since it uses the expensive I/O to read and rewrite the whole data-set. In the practical server setting, there are some opportunities to mitigate the costly shuffle: 1) Perform the shuffle during the off-line time. 2) Considering the client-and-server setting, the shuffle only runs on the remote server, so there is no need to transmit data over the slow network. 3) As shown in our experimental results, the shuffle process consists of sequentially read and write operations, which is 10 times faster than the random data access to the disk. For the ideal case, without considering the shuffle as an extra overhead, our H-ORAM can theoretically achieve 32 times faster access time than the Path ORAM.

## 5.2 Experimental Results

**Experiment Setup:** We implement our ORAM interface in a real machine by using the configurations in Table 2. We use HDD as our storage backend, with the average read/write throughput shown below. We implement Path ORAM and H-ORAM with the naive setting (no recursive). The file size is set to 1KB. We test our interface through both small dataset(25, 000 requests, 64MB storage) and large dataset.(500, 000 requests, 1GB storage)

**Table 2: Experimental Machine Setup**

| | |
|---|---|
| **Operating system** | Linux Ubuntu 16.4 |
| **CPU** | Intel i7-7700K |
| **Memory** | DDR4 Pc4-2133 16GB |
| **Disk** | HDD 7200 RPM 500GB |
| **Read/Write Throughput** | 102.7 MB/s, 55.2 MB/s |

**Table 3: Test cases characteristics and results**

| | Locality | $\bar{c}$ | # of REQ | # of I/O | Effective hit% |
|---|---|---|---|---|---|
| **webserver** | 0.13 | 4 | 25,000 | 6,539 | 95.5% |
| **oltp** | 0.07 | 4 | 25,000 | 6,357 | 98.4% |
| **varmail** | 0.015 | 4 | 25,000 | 6,344 | 98.4% |
| **webserver** | 0.13 | 4 | 500,000 | 131,210 | 95.2% |

**Locality analysis:** As is shown in section 4.2, the H-ORAM interface requires users to adjust the cache degree by the data locality, or, in other word, the distribution of access pattern. In this section, we test our interface with several workloads from FileBench[16]. Table 3 shows the characteristics of our test cases. As we defined at Section 4.2, $\bar{c}$ is the average preset cache degree, the number of in-memory access during the I/O transmitting. The value of locality is calculated by the ratio of data that has been accessed over the total number of data in the storage. We preset a conservative value $\bar{c}=4$ for all the test case to ensure the obliviousness. If all the requests hit in the memory, the ideal number of I/O accesses is $25,000/4=6250$. The effective hit rate show the actual cache effect of our design, which is computed as the ideal number (6, 250) of access over the actual number of I/O access (see Table 3. # of I/O).

**Table 4: 64 MB data set with 25,000 requests**

| | H-ORAM | Path ORAM |
|---|---|---|
| **Storage/Memory Size** | 64 MB/8MB | 120 MB / 8MB |
| **Number of I/O Accesses** | 6,539 | 25,000 |
| **I/O latency** | 77 $\mu s$ | 1,032 $\mu s$ |
| **Shuffle time** | 729 ms × 1 | N/A |
| **Total time** | 1,232 ms | 25,575 ms |

**Table 5: 1 GB data size with 500,000 requests**

| | H-ORAM | Path ORAM |
|---|---|---|
| **Storage/Memory Size** | 1 GB/ 128 MB | 1.875 GB / 128 MB |
| **Number of I/O Accesses** | 131,210 | 500,000 |
| **I/O latency** | 107 $\mu s$ | 1,364 $\mu s$ |
| **Shuffle time** | 9,743 ms × 2 | N/A |
| **Total Time** | 33,525 ms | 682,041 ms |

In addition, we show two sets of results that represent small and large size of data set in Table 4 and 5.

The experimental results show the I/O latency for one ORAM access is reduced by approximately 13 times. The performance

gain mainly comes from the reduction of I/O accesses summarized below:

$$\text{H-ORAM :1page (Read)}$$
$$\text{path ORAM :4pages (Read) + 4pages (Write)}$$

For the system setting in the table 2, the tested HDD backend has as a read speed twice faster than the write. As the result, the theoretical gains for I/O latency is $4 + 4 \times 2 = 12$ times of H-ORAM, which is closed to the measured improvement (13×).

In addition, we observe that the shuffle speed is much faster than the theoretical calculation due to the intrinsic properties of HDD. The access speed is greatly depends on the randomness of requests. When the Path ORAM accessing 4 buckets, it needs to go through 4 sparse locations to fetch the data. For example, $\{4161, 41090, 114948, 262665\}$ are 4 bucket addresses when the Path ORAM access the leaf 33289. We observe a large variance between these four addresses, which adds extra overhead when using the HDD. On the other hand, for the H-ORAM, during the shuffle process, the whole data set is sequentially loaded and written, and it can benefit from the fast sequential access speed of HDD devices, which is 10× to 20× faster than the random page reading.

### 5.3 Discussion on optimization

After years of exploration, numerous of optimization methods have been developed for the Path ORAM and this trend will continue. Our proposed H-ORAM, also assembly the Path ORAM on the top level and optimize the protocol to suit larger data set with a cacheable interface design. The previous studies, that aims at optimizing the position map, stash or tree-top data can also be applied to our H-ORAM. In this section, we discuss a few potential optimization that can be build on H-ORAM to improve the performance.

**Partial shuffle:** As shown in the experimental results, we reduced the ORAM I/O overhead to a minimal amount. The most significant time spent on H-ORAM, is the shuffle period, which happen every $n$ I/O operations. A full shuffle of entire data is costly, therefore, we propose a lightweight and flexible partial shuffle protocol. For every shuffle period, we only need to shuffle a portion of the data, for example, $r = 1/4N$. Instead of shuffling each partition every period, one partition is going to shuffle every 4 periods. The evicted data from memory keep concatenating on the top of each partition until 4 period after last shuffled. With the partial shuffle, we need to issue oblivious access that touches more redundant data each time. The less we shuffle, the more redundant accesses are required. Through this method, we can compute a proper shuffle ratio with a system profiling, which balances the shuffle overhead and the I/O overhead.

**Multi-users Case:** Another use case is that, when the ORAM protected data set is shared by multiple users, the system needs to provide high throughput while maintain the obliviousness access between different users. Our proposed H-ORAM groups multiple requests in the scheduler, to maximize the memory bandwidth. When there are multiple users, we can continue to use the group strategy so that requests from different users can be issued at the same time. To protect the access pattern from potential malicious users, some access control protection is required and can be added to our scheduler.

## 6 CONCLUSIONS

Current ORAM designs such as square root ORAM, Path ORAM, face the challenges when the data set grows out of the capacity of main memory. The unavoidable I/O accesses brings extra overhead to the expensive protocols. Meanwhile, it is hard to cache the ORAM accesses because of the unique memory organization. In this work, we propose a novel cacheable ORAM interface, H-ORAM, to reduce the I/O access overhead per ORAM access, and the reshuffle overhead which happens on background. In our theoretical and experimental results, we show that our proposed H-ORAM outperforms the state-of-the-art Path ORAM by 19.8 times for a small data set and 22.9 times for a large data set.

## REFERENCES


[1] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. 2002. The EM side-channel(s). In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 29–45.

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious File System for Intel SGX. (2018).

[3] Sundeep Bajikar. 2002. Trusted platform module (tpm) based security on notebook pcs-white paper. *Mobile Platforms Group Intel Corporation* (2002), 1–20.

[4] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 182–194.

[5] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[6] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.. In *Ndss*, Vol. 20. 12.

[7] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel® Software Guard Extensions: EPID Provisioning and Attestation Services. *White Paper* 1 (2016), 1–10.

[8] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices* 35, 11 (2000), 168–177.

[9] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. 2014. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming*. Springer, 556–567.

[10] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2017. Cacheshuffle: An oblivious shuffle algorithm using caches. *arXiv preprint arXiv:1705.07069* (2017).

[11] Ling Ren, Christopher W Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2017. Design and Implementation of the Ascend Secure Processor. *IEEE Transactions on Dependable and Secure Computing* (2017).

[12] Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten Van Dijk, and Srinivas Devadas. 2013. Design space exploration and optimization of path oblivious ram in secure processors. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 571–582.

[13] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. 2017. ZeroTrace: Oblivious memory primitives from Intel SGX. In *Symposium on Network and Distributed System Security (NDSS)*.

[14] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 253–267.

[15] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 299–310.

[16] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *login: The USENIX Magazine* 41, 1 (2016).

[17] Rujia Wang, Youtao Zhang, and Jun Yang. 2017. Cooperative Path-ORAM for Effective Memory Bandwidth Sharing in Server Settings. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 325–336.

[18] Zhenghong Wang and Ruby B Lee. 2006. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 473–482.

[19] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting square-root ORAM: efficient random access in multi-party computation. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 218–234.