Jump-ORAM: An ORAM Scheme Achieving Constant Bandwidth Blowup Without Server-side Computing Overhead

***Abstract*—The ORAM is a cryptographic protocol that hides client access patterns from leaking. Prior works suffer the cost of logarithmic communication blowup or server-side computing overhead. In this paper, we present an ORAM scheme named Jump-ORAM that achieves constant bandwidth blowup without server-side computing overhead. To achieve above goal, we map a request for one block into the corresponding request for a given number of blocks. To implement above mapping, we first propose a data structure named position map to convert the accessed block's logical-ID into its physical-ID, and then design a selection algorithm to map the accessed block's physical-ID to a given number of physical-IDs. To ensure the security of the Jump-ORAM, we periodically and randomly rewrite accessed blocks back to the server without leaking their new assigned physical-IDs. Specifically, we propose a data structure named the data cache to temporarily store accessed blocks. We design a swap algorithm to rewrite accessed blocks back to the server without leaking their new assigned physical-IDs. We proved that our scheme is secure under a statistical model. We compared Jump-ORAM's performance with its counterparts. Experimental results demonstrate the efficiency of Jump-ORAM. It is approximately $100\times$ and $633\times$ faster than Path-ORAM and S^3ORAM, respectively.**

***Index Terms*—Access Pattern, ORAM, Constant Bandwidth Overhead, Privacy Preserving**

I. Introduction

A. Motivation and Problem Statement

With the growing popularity of cloud storage and its convenience, more and more people are storing encrypted data on remote servers. However, their data privacy is also under a new threat. The adversary (*e.g.* server) could statistically deduce some sensitive information by correlating the sequences of storage positions accessed by a client (*e.g.*, access patterns) with other public information he could have. For example, Pinkas *et al.* pointed out that attackers could infer the correlation between access patterns and the timing of stock transactions by observing financial data [3]. Islam *et al.* showed that attackers could infer 80% of the query content by monitoring the mailbox data [4].

Oblivious Random Access Machine (ORAM) is a cryptographic protocol that allows a client to access her encrypted data on a remote server without leaking her access patterns. Specifically, after implementing an ORAM scheme, adversaries cannot know (1) which block is being accessed, (2) whether the same block is being accessed, (3) whether an access operation is reading or updating. Unlikely, in the early days of ORAM, it was considered a theoretical solution instead of a practical one because of its worst-case linear bandwidth overhead [3], [5]. Decreasing the bandwidth overhead is therefore of significance in putting the ORAM into practice. In this paper, we aim to design a secure ORAM scheme that achieves both constant bandwidth overhead and zero server-side computing overhead.

B. Limitation of Prior Art

No prior works simultaneously achieve $O(1)$ bandwidth blowup and zero server-side computing overhead. The works [6], [7] reduce ORAM's bandwidth overhead by optimizing server's storage structures without server-side computing overhead. However, they suffer the $\Omega(logN)$ bandwidth overhead. The works [1], [8], [9] use server-side computing overhead to achieve the constant bandwidth overhead. The computational cost introduced to reduce the bandwidth overhead is even greater than the logarithmic communication blowup. Hoang *et al.* present that the efficiency of Path-ORAM is three orders of magnitude more than Onion-ORAM [1]. Note that the Onion-ORAM [8] achieves $O(1)$ bandwidth blowup with server-side computing overhead and Path-ORAM [2] achieves $O(logN)$ bandwidth blowup without server-side computing overhead.

C. Adversary Model

We assume that the server is semi-honest (honest-but-curious), which has been adopted by prior works [1], [5], [8]–[20]. This assumption means that the server will not intentionally tamper with or modify the encrypted data but may be curious to incur some sensitive information from the client's access patterns.

D. Our Approach

In this paper, we propose a secure ORAM scheme named Jump-ORAM that achieves constant bandwidth blowup without server-side computing overhead. To achieve constant bandwidth blowup without server-side computing overhead and hide which block is being accessed by the client, we map a client's request for one block into the corresponding one for a given number of blocks. To implement the above mapping, we first propose a data structure named position map to convert the accessed block's logical-ID into its physical-ID, where the logical-ID and physical-ID represent the block index

TABLE I
Asymptotic and Experimental Performance Comparison of Jump-ORAM and its Counterparts.

Scheme	Bandwidth overhead	Server's computation overhead	Initialization delay(s)	Access delay(s)
S^3ORAM [1]	$O(1)$	✓	4074	38
Path-ORAM [2]	$O(logN)$	✕	6391	6
Non-Secure Baseline	$O(1)$	✕	565	0.03
Jump-ORAM	$O(1)$	✕	758	0.06

This table provides the experimental result of the Jump-ORAM with its counterparts with 20GB database containing 512-KB blocks. We refer the reader to Section VI for the details of our experiments.
✓ and ✕ denote the scheme utilizing the server-side computation overhead and not, respectively.

on the client-side and server-side, respectively. Second, we present a selection algorithm to map the accessed block's physical-ID into a given number of physical-IDs, the primary technique for which is an iterative formula. To meet the requirements of the iterative formula, we let the server use an array data structure to store encrypted data. To ensure adversaries cannot know whether the same block is being accessed, we need to randomly and periodically rewrite accessed blocks back to the server without leaking their new assigned physical-IDs. To achieve this goal, we propose a data structure named data cache to store the accessed blocks temporarily. With the number of access times increase, the data cache with a fixed size will overflow. To address the overflow problem, we periodically rewrite accessed blocks in the data cache back to the server without leaking their new assigned physical-IDs. Therefore, we present a swap algorithm to secretly and periodically rewrite the accessed blocks back to the server. To hide whether an access operation is reading or updating, we rewrite the re-encrypted blocks in the stash back to the server per the client's access. Table I outlines a comparison of Jump-ORAM and its counterparts.

E. Our Contributions

Our contributions are as follows. (a) We propose an ORAM scheme named Jump-ORAM achieves constant bandwidth blowup without server-side computing overhead. (b) We first use an iterative formula based on the server's data structure array to construct the mapping from one block to more blocks. (c) We proved the security of our scheme under a statistical security model. (d) We conducted comprehensive experiments to evaluate the performances of the Jump-ORAM and its counterparts including Path-ORAM and S^3ORAM.

II. Related Work

ORAM schemes achieving $\Omega(logN)$ bandwidth blowup without server-side computing overhead: Goldreich and Ostrovsky propose Trivial-ORAM, Square-Root-ORAM, and Hierarchical-ORAM that achieve the $O(N)$, $O(\sqrt{N}log^2N)$, and $O(log^3N)$ amortized bandwidth blowup, respectively [6]. Despite rich works [3], [10], [13], [14], [21]–[24] improving the performance of Square Root-ORAM and Hierarchical-ORAM, they cannot hit the optimal lower bound of $O(logN)$ bandwidth blowup [25]–[27] in worst-case condition. The major reason for this is the expensive oblivious sorting operations required by hierarchical-based framework. Shi *et al.* break the hierarchical-based paradigm, and propose Tree-ORAM based on binary-tree construction [7]. Unfortunately, the Tree-ORAM suffers $O(log^3N)$ worst-case performance. Subsequent works [2], [11], [28]–[31] strive to improve the practical performance of the ORAM based on the binary-tree framework. The Path-ORAM [2] distinguish itself among these works due to its $O(logN)$ bandwidth bandwidth. However, it is still costly for certain applications due to its logarithmic communication overhead for each access request [12], [15]–[17].

ORAM schemes achieving $O(1)$ bandwidth blowup with server-side computing overhead: Apon *et al.* show that ORAM schemes with Fully Homomorphic Encryption (FHE) could obtain constant communication overhead [32]. But its techniques such as bootstrapping and SNARKs require non-standard assumptions. Onion-ORAM [8] leverages poly-logarithmic server computation (Additively Homomorphic Encryption(AHE)) to obtain the constant worst-case bandwidth blowup under standard assumptions. In spite of the constant bandwidth cost the Onion-ORAM has achieved, it suffers even more delay owing to costly homomorphic operations [1], [33]. Subsequent works [1], [34]–[36] focus on optimizing the expensive homomorphic operations by multi-servers, Shamir Secret Sharing(SSS), multi-server PIR and XOR operations. Abraham *et al.* show that any ORAM scheme with PIR is restricted to an asymptotically tight sub-logarithmic bandwidth blowup bound of $\Omega(log_{cD}N)$ [37], where c,D represent the numbers of blocks and operations performed by PIR, respectively. There exists security flaws in CH^f-ORAM [34] and C-ORAM [35].

TABLE II
NOTATIONS

Symbol	Description
logical-ID	*The index of blocks from the view of the server or data cache*
physical-ID	*The index of blocks from the client's view*
N	*The number of real blocks outsourced to the server*
N_{sto}	*The number of real and dummy blocks outsourced to the server*
DC	*The data structure of data cache in the client*
S	*The data structure of stash in the client*
$Pmap$	*The data structure of position map in the client*
$x \leftarrow Pmap[a]$	*x represents the physical-ID of the block a*
$seq(a)$	*The index sequence of block a calculated by Algorithm 1*

III. The Jump-ORAM Protocol

We first introduce the layout of this section. The Jump-ORAM's description begins with its overview, followed by its data structures and algorithms, the initialization and finally, the Jump-ORAM access protocol. We describe the notations used in this paper in Table II.

A. Overview

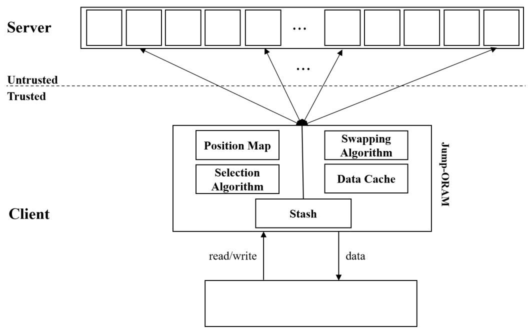


Fig. 1. The Jump-ORAM Overview

We consider a scenario in Figure 1 where a client accesses her encrypted data in a remote server without leaking her access patterns via the Jump-ORAM. Specifically, first, we map the client's request for a block into the one for $S.size$ blocks via a position map and selection algorithm, where $S.size$ denotes the stash size. The function of the position map and selection algorithm is to convert the accessed block's logical-ID to its physical-ID and to map the accessed block's physical-ID into $S.size$ physical-IDs, respectively. Second, the server sends $S.size$ blocks corresponding to the request of Jump-ORAM to the stash. Third, we find the client's block of interest by traversing the blocks in the stash. Fourth, we swap the client's block of interest with a block in the data cache according to the rules of the swap algorithm. Last, we rewrite blocks in the stash back to the server and send the data to the client.

B. Data Structure and Algorithm

Data Structure in Server-side: As shown in Figure 1, the server uses an array (denoted as A) to store the encrypted data outsourced from the client and each element in A is the block. Note that to satisfy the requirements of selection algorithm, we let $A.size = k \times S.size$, where $k \in N^*$.

Data Structure in Client-side: As shown in Figure 1, the data structures of Jump-ORAM in client-side contain a position map, data cache, and stash. The position map is a hash table listing real blocks' corresponding physical-IDs. The position map updates each time after client's access. According to the above description, we describe the size of position map as $Pmap.size = r \times N$, where $r \in (0,1)$. The data cache is an array temporarily storing the blocks accessed by the client. To ensure the rewriting the blocks in the data cache to the server randomly and secretly, we let $DC.size \geq 2, DC.size \in N^*$, where $DC.size$ denotes the size of the data cache. In client's each access, the stash temporarily stores the blocks retrieved from the server. To ensure the server cannot know which block is being accessed, we let $S.size \geq 2, S.size \in N^*$, where $S.size$ denotes the stash size. Note that $S.size$ is equal to the Jump-ORAM's bandwidth overhead, which we will analyze in Section IV.

Algorithm 1: Selection

```
Input: $x, S.size, N_{sto}$
Output: $S$
1 $\bar{x} \leftarrow x$;
2 $stepLen \leftarrow N_{sto}/S.size$;
3 do
4     $x \leftarrow (x + stepLen) \mod N_{sto}$;
5     $X.append(x)$;
6 while $x \neq \bar{x}$;
7 $X.sort$;
8 $S \leftarrow$ fetch the block $x \in X$ and decrypt it;
9 return $S$;
```

Selection Algorithm: We take a block's physical-ID (x), the size of stash ($S.size$) and the total number (N_{sto}) as the selection algorithm's inputs. Furthermore, its output is the stash (S) filled with blocks fetched from the server. The conversion from input to output is as follows. First, we get $stepLen$ by formula $stepLen = N_{sto}/S.size$. Second, we append a physical-ID sequence to the physical ID set X. Note that the physical-ID sequence is calculated by a iterative formula $x_{i+1} = (x_i + stepLen) \mod N_{sto}$, where $i \in [1, S.size - 1]$ and $x_1 = x$. Third, we sort X in ascending or descending order. Last, we fetch the blocks denoted by $x \in X$ from the server to the stash and decrypt them. We show the pseudo-code of selection algorithm in Algorithm 1.

Swap Algorithm: We take a physical-ID (x), the stash (S), data cache (DC) and position map ($Pmap$) as The swap algorithm's inputs. Moreover, its output is a random integer denoted by $\bar{x} \in [0, N_{sto} - 1]$ or updated S, DC, and $Pmap$ depending on where the physical-ID x is. Specifically, if block x is in the data cache, the return of swap algorithm is a random integer $\bar{x} \in (0, N_{sto} - 1)$. In addition, if block x is in the server, we randomly swap the block denoted by x in the stash with a block in the

Algorithm 2: Swap

```
Input: $x, S, DC$, and $Pmap$
Output: $\bar{x}$ or ($S, DC$, and $Pmap$)
1  if $x \in DC$ then
2      $\bar{x} \leftarrow$ UniformRandom$(0, N_{sto} - 1)$;
3      return $\bar{x}$;
4  else
5      $\tilde{x} \leftarrow$ UniformRandom$(0, DC.size - 1)$;
6      $DC \leftarrow (DC - \{\tilde{x}, data\}) \cup \{x, data\}$;
7      $S \leftarrow (S - \{x, data\}) \cup \{\tilde{x}, data\}$;
8      update$(Pmap)$;
9      return $S, DC$, and $Pmap$;
10 end
11 return;
```

data cache and updates the position map via the swap algorithm. In other words, if block x is in the server, the returns of the swap algorithm are updated stash, data cache, and position map. We describe the pseudo-code of the swapping algorithm in Algorithm 2.

C. The Jump-ORAM Initialization

We takes client's database (DB), empty position map (Pmap) and the size of stash (S.size) as the initialization protocol's inputs. Furthermore, its output is updated position map. The conversion from input to output is as follows. First, we organize client's database into N blocks called real blocks and tag them with corresponding logical-IDs. Second, we get the total number of blocks N_{sto} in the server, according to the selection algorithm, $A.size = N_{sto}$. Third, we use the random integers ranging from 0 to $N_{sto} - 1$ to generate the position map. Fourth, we write the encrypted blocks into the corresponding position of A based on the position map. Fifth, we fill up the rest space of A by the encrypted dummy blocks. Finally, we send A to the remote server. We present its pseudo-code in Protocol 1.

Protocol 1: JumpORAM.init()

```
Input: $DB, Pmap,$ and $S.size$
Output: $Pmap$
1  organize DB into blocks$(b_0, \cdots, b_{N-1})$
     with logical-IDs$(id_0, \cdots, id_{N-1})$;
2  $N_{\text{sto}} \leftarrow \lceil N/S.size \rceil \times S.size$;
3  for $i = 0, \cdots, N-1$ do
4      Pmap$[id_i] \leftarrow$ UniformRandom$(0, N_{sto} - 1)$;
5      encrypt the block $b_i$;
6      write block $b_i$ into Pmap$[id_i]$ position of $A$;
7  end
8  $generate(N_{\text{sto}} - N)$ dummy blocks ;
9  fill rest space in $A$ by encrypted dummy blocks;
10 send the array $A$ to the remote server;
11 return $Pmap$ ;
```

D. The Jump-ORAM Access Protocol

We take the operation of reading or updating (op), block of interest's logical-ID (a) and new data ($data^*$) as the access protocol's inputs. Furthermore, its output is the decrypted client's block of interest. The conversion from input to output is as follows. First, we map the client's block of interest a into its physical-ID x based on the position map. Second, we get where the block a is according to the x. If the block a is in the data cache, we randomly access a block in the server. We then fetch the block a from the data cache and send it to the client. We renew the corresponding block in the data cache, if the client's operation updating. On the flip side, If the block a is in the server, we call the selection function to fetch $S.size$ blocks from the server to the stash. We then send the decrypted block a to the client. We renew the corresponding block in the data cache, if the client's operation updating. We last call the swap function to randomly swap the block a with a block in the data cache. Finally, we send the re-encrypted blocks in the stash back to the server. We describe the pseudo-code of Jump-ORAM's access protocol in Protocol 2.

Protocol 2: JumpORAM.access()

```
Input: $op, a,$ and $data^*$
Output: $data$
1  $x \leftarrow$ Pmap$[a]$;
2  if $x \in DC$ then
3      $\tilde{x} \leftarrow$ swap$(x, S, DC)$;
4      selection$(\tilde{x}, S, N_{sto})$;
5      $data \leftarrow$ read block $a$ from DC;
6      if op = update then
7          $DC \leftarrow (DC - \{x, data\}) \cup \{x, data*\}$;
8      end
9      $\bar{x} \leftarrow$ UniformRandom$(0, N_{sto} - 1)$;
10     return $\bar{x}$;
11 else
12     $S =$ selection$(x, S, N_{sto})$;
13     $data \leftarrow$ read block $a$ from DC;
14     if op = update then
15         $S \leftarrow (S - \{x, data\}) \cup \{x, data*\}$;
16     end
17     swap$(x, S, DC)$;
18 end
19 send re-encrypted blocks in the $S$ to the server;
20 return $data$;
```

In Protocol 2, if the client reads block a, the protocol can be described as following.

$$data \leftarrow \textbf{JumpORAM.access}(read, a, None) \quad (1)$$

Moreover,if the client updates block a, the protocol can be described as following.

$$data \leftarrow \textbf{JumpORAM.access}(update, a, data^*) \quad (2)$$

We next present an example to illustrate the access protocol further. Suppose a client want to access block $a = 1$. First, we convert the logical-ID 1 to its physical-ID (*e.g.*, 4) based on the position map. Second, we check out where the block a is. Suppose the block a is in the server. Third, we use the selection function to retrieve the blocks 4,7,and 1 from the server to the stash with $N_{sto} = 9$, $S.size = 3$. Fourth, if $op = read$, we select the client's block of interest from the stash. Otherwise, we use the $data^*$ to update the corresponding block in the stash. Fifth, we use the swap function to randomly swap the block a with a block in the data cache. Last, we send re-encrypted blocks in the stash to the remote server.

IV. Security Analysis

We use the security definition for ORAMs adopted by [2], [5], [38] to prove the security of Jump-ORAM. An ORAM scheme is secure if it satisfies the conditions. The first condition (obliviousness) is that the server cannot distinguish $A(\vec{y})$ and $A(\vec{z})$ under $\vec{y}.size = \vec{z}.size$ after implementing the ORAM scheme, where $\vec{y}$ and $\vec{z}$ are the logical-ID sequences of size M requested by the client. Note that from the Section III, the server only see the $A(\vec{y})$, a sequence corresponding to $\vec{y}$, composed by physical-ID sequences. We describe the $A(\vec{y})$ as follows.

$$A(\vec{y}) = (seq(a_M), seq(a_{(M-1)}), \cdots, seq(a_1)) \tag{3}$$

where $seq(a_i)$ represents physical-ID sequence corresponding to the block a_i for the $i_$th access operation. The second condition (correctness) is that the ORAM's returns on input $\vec{y}$ is consistent with $\vec{y}$ with probability ≥ 1 - $negl(\vec{y}.size)$.

Theorem 1 (Security): The Jump-ORAM is statistically secure.

Proof. We first prove the obliviousness of Jump-ORAM. If the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ are statistically independent for $i < j$, we can use the Multiplication Rule for Independent Events to get the Jump-ORAM's probability that the server correctly targets the $\vec{y}$ from the $A(\vec{y})$. We describe the probability as follows.

$$\mathrm{Pr}_{\mathrm{jump}}(A(\vec{y})) = \prod_{i=1}^{M} \mathrm{Pr}_{\mathrm{jump}}(\mathrm{seq}(a_i)) \tag{4}$$

$$\mathrm{Pr}_{\mathrm{jump}}(\mathrm{seq}(a_i)) = \frac{N_{sto}}{N_{\mathrm{sto}} + DC.size} \cdot \frac{S.size}{N_{sto}}, \ \mathrm{i} \in [1, M] \tag{5}$$

where $DC.size \geq 2$ and $N_{sto} >> S.size$, which means the $\mathrm{Pr}_{\mathrm{jump}}(\mathrm{seq}(a_i)) << 1$ and negatively related with N_{sto}. This proves that the server cannot distinguish $A(\vec{y})$ and a random sequence of bit strings.

We now prove the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ are statistically independent for $i < j$. (1) For any logical-ID a_i and a_j, $i < j$ in the access sequence, if the block a_i or a_j is in data cache, the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ are statistically independent, where $seq(a_i)$ ($seq(a_j)$) represents physical-ID sequence corresponding to the block a_i (a_j) for the $i_$th ($j_$th) access operation. Because according to the Algorithm 2, the Jump-ORAM accesses a random block when the accessed block is in the data cache. (2) If $a_i = a_j$, $j = i + 1$ in the access sequence, the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ are statistically independent. From the condition $a_i = a_j$, $j = i + 1$, we know that the block a_j must be in the data cache. Therefore, the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ are statistically independent under the condition $a_i = a_j$, $j = i + 1$. (3) If $a_i = a_j$, $i + 1 < j$ in the access sequence, the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ are statistically independent. Case 1: If the block a_i or a_j is in the data cache, the $\mathrm{seq}(a_i)$ is statistically independent of $\mathrm{seq}(a_j)$. Case 2: If the block a_i and a_j are both not in the data cache, the block a_i can be in any physical-ID in the server according to the Algorithm 2., Therefore, the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ are statistically independent with the condition $a_i = a_j$, $i + 1 < j$. (4) If $a_i \neq a_j$, $i + 1 < j$ in the access sequence, the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ are statistically independent. Case 1: If the block a_i or a_j is in the data cache, the $seq(a_i)$ is statistically independent of $seq(a_j)$. Case 2: If the block a_i and a_j are both not in the data cache, the $\mathrm{seq}(a_i)$ is no related with $\mathrm{seq}(a_j)$ according to the Algorithm 2, which means the $\mathrm{seq}(a_i)$ is statistically independent of $\mathrm{seq}(a_j)$. Therefore, the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ are statistically independent with the condition $a_i \neq a_j$, $i + 1 < j$. In conclusion, the $\mathrm{seq}(a_i)$ and $\mathrm{seq}(a_j)$ for $i < j$ are statistically independent.

We second prove the correctness of Jump-ORAM. According to the Algorithm 1, when client's block of interest a is in the server, the Jump-ORAM use the physical-ID of the block a to get the $\mathrm{seq}(a)$. In other words, if the block a is in the server, the corresponding $S.size$ blocks retrieved from the server must contain the block a . On the flip side, if the block a is in the data cache, the Jump-ORAM directly return the block a to the client from the data cache. In conclusion, the Jump-ORAM always return the client's block of interest to the client correctly. In other words, the Jump-ORAM achieves the correctness.

V. Asymptotic Cost Analysis and Optimization

We first describe the Jump-ORAM's asymptotic cost of bandwidth overhead, server storage, and client storage. Then introduce two optimization techniques to improve the Jump-ORAM's performance.

A. Asymptotic Cost Analysis

There are preliminary concepts used latterly. The bandwidth overhead/blowup refers to the ratio of the number of blocks the ORAM scheme needs to access to the number of data blocks that need to be accessed without the ORAM per client's access. Without loss of generality, we suppose the scheme without ORAM needs to access a block twice for each access by the client. We use the space complexity with block as a unit to illustrate the asymptotic cost for server and client storage.

Achieving constant bandwidth blowup without server-side computing overhead: According to Protocol

2, the Jump-ORAM retrieves the $S.size$ blocks from the server and returns the re-encrypted $S.size$ blocks to the server, per the client's access. According to the above description of bandwidth overhead, we get that Jump-ORAM's bandwidth overhead is $S.size$ a fixed integer. Therefore, we conclude that the Jump-ORAM achieves the $O(1)$ bandwidth overhead. Moreover, the server only serves as a storage device, which means Jump-ORAM has no need of the server-side computing overhead.

Achieving $O(N)$ server storage and $O(N/R)$ client storage: According to the Protocol 1, the number of blocks in the server is $N_{sto} = \lceil N/S.size \rceil \times S.size$. If $N\%S.size = 0$, we get $N_{sto} = N$. Otherwise, $N_{sto} = N + (S.size - (N\%S.size)), 1 \leq (N\%S.size) < S.size$. Owing to the S.size being a fixed number and irrelated with N, the Jump-ORAM achieve $O(N)$ server storage. From the Section III-B, the client's data structure was composed of the position map, data cache, and stash. The size of the data cache and stash is fixed. Therefore, we only consider the size of the position map. In practice, we usually implement the position map by the data structure of the array with unsigned int as its element. Stefanov *et al.* refer that the block size ranges from 64 to 256 KB [2]. We define R as the ratio of block size to the element size in the array and it ranges from 8192 to 32768. Therefore, the Jump-ORAM achieve the $O(N/R)$ client storage. In addition, the size of the position map is usually a small value according to the R's range.

B. Optimization

Increasing the security via flexible stepLen: We get the physical ID sequence according to the Algorithm 1 with the fixed *stepLen*. The method under the fixed *stepLen* decreases the total number of sequences from N_{sto} to $N_{sto}/S.size$. Furthermore, this method depresses the security of the Jump-ORAM, which can be indicated by Equation 5 in Section IV. We propose a flexible *stepLen* method to increase the security of the Jump-ORAM. What is flexible *stepLen* method? As the name suggests, the *stepLen* refers to the *stepLen* in selecting the physical IDs is flexible. We use the following iterative formula to describe it.

$$x_{i+1} = (x_i + rand) \bmod N_{\text{sto}} \tag{6}$$

where $1 \leq x_i \leq N_{\text{sto}}, 1 \leq rand, i < N_{\text{sto}}$, x_i and $rand$ represent the physical ID of a block and a random integer ranges from 1 to N_{sto}, respectively. We use the following example to illustrate flexible *stepLen* technique. Suppose the client wants to fetch a block its physical ID 4 with $N_{sto} = 9$ and $S.size = 3$. Firstly, we get a $rand = 3$ and the second element in the sequence is 7. Secondly, we get another $rand = 5$ and the third element in the sequence is 3. Finally, we get the physical ID sequence $(4,7,3)$. From the Combination view, we get all possible $S.size$ combinations from N_{sto} by flexible *stepLen* technique. Therefore, We rewrite the Equation 5 as followings.

$$\Pr_{jump}(eq(a_i)) = \frac{N_{\text{sto}}}{N_{\text{sto}} + DC.\text{ size}} \cdot \frac{1}{C_{N_{sto}}^{\text{S.size}}} \tag{7}$$

where $C_{N_{Sto}}^{\text{S.size}}$ represents the combination formula of N_{sto} choosing $S.size$. According to the unimodality of binominal coefficients, we get $C_{N_{Sto}}^{\text{S.size}} \geq C_{N_{\text{sto}}}^{1} = N_{\text{sto}} > \frac{N_{\text{sto}}}{\text{S.size}}$, under $S.size > 1$. In conclusion, the flexible *stepLen* technique increases the security of the Jump-ORAM.

Achieving the $O(1)$ client storage via recursion: As mentioned in the Section V-A, the size of the position map determines the asymptotic cost of client storage. Therefore, we can achieve the $O(1)$ client storage by decreasing the size of the position map of Jump-ORAM in the client. We leverage the recursion technique adopted by prior works [1], [2], [5], [38] to deduce the space occupied the position map. The recursion technique uses the recursive mindset to view the data in the position map as data blocks transferred to the remote server. Specifically, we construct a series of ORAMs called $ORAM_0, ORAM_1, \cdots, ORAM_X$, where $ORAM_0$ stores the data blocks, the $ORAM_{(i+1)}$ contains the position map in the $ORAM_i$, and the client only conserves the position map of the $ORAM_X$.

We illustrate the costs that the Jump-ORAM pays for achieving the $O(1)$ client storage. Without loss of generality, we suppose the size of position map of the $ORAM_X$ is one block. We get the asymptotic value of X is $O(logN)$. For Jump-ORAM in recursive version, if the client wants to access a block in $ORAM_0$, the Jump-ORAM needs to find its physical ID in $ORAM_1$, which triggers a recursive call to find the physical ID of the physical ID in $ORAM_2$ and so on until find the corresponding physical ID in $ORAM_X$. In other words, the Jump-ORAM sacrifices the $O(logN)$ communication *rounds* to achieve the constant client storage complexity.

VI. Experimental Evaluation

We first describe the evaluation metrics and methodology. We then present our implementation configuration and details, followed by the comparison of Jump-ORAM and its counterparts under the same running environment. Note that in this experimental evaluation, we evaluate all ORAM schemes under their non-recursive form, where the client stores the position map locally.

A. Evaluation Metrics and Methodology

Evaluation metrics: We compared Jump-ORAM with its counterparts based on: (1) initialization delay for building ORAM structure; (2) access delay for accessing a client's block of interest since the former influences the first impression of the ORAM scheme for users and the latter impacts the user experience during the ORAM running. We also analyzed the breakdown cost of initialization and access delay to find the factor that most significantly influences them. Specifically, we broke the initialization delay into (1) building the metadata (*e.g.* position map), (2) creating the database, (3) sending the database to the server. We broke the access delay into (1) client computation, (2) client-server communication, (3) disk I/O access time, and ((4) server computation).

We chose Path-ORAM, S^3ORAM, and Non-Secure Baseline, an insecure encrypted data storage scheme we designed in this paper, as Jump-ORAM's counterparts. Because the Path-ORAM outperforms the ORAM schemes that achieve $\Omega(logN)$ bandwidth blowup without server-side computing overhead. The S^3ORAM distinguishes itself from the ORAM schemes achieving $O(1)$ bandwidth blowup with server-side computing overhead. The access delay of Non-Secure Baseline is the lowest delay benchmark for any ORAM scheme, which can tell us the Jump-ORAM's room for improvement.

Evaluation methodology: We describe our methodologies as follows.

- Jump-ORAM: We implemented the Jump-ORAM with a fixed *stepLen*. We selected the size of the stash and data cache as $S.size = DC.size = 2$ to reduce the execution time of Jump-ORAM as much as possible under its security condition.
- Path-ORAM: We selected the size of bucket as 5 to ensure a negligible stash overflow probability of 2^{-80}. We let each access operation be followed by an eviction operation to avoid the bucket overflow problem. We measured the path-ORAM access delay taking into account the cost of the download and upload blocks and the amortized cost of eviction.
- S^3ORAM: We set the value of eviction rate and the size of the bucket as 333 and 333, respectively, to obtain a negligible stash overflow probability of 2^{-80}. And, we chose the number of servers and the privacy level as 3 and 1, respectively, to shorten the execution time of S^3ORAM. Note that S^3ORAM with privacy level 1 cannot be applied in real-world scenarios because any server has the ability to decrypt encrypted data stored on the server under the condition where privacy level is equal to 1. We gauged the access delay of S^3ORAM considering the cost of the download and upload shares and the amortized cost of eviction.
- Non-Secure Baseline: We implemented the Non-Secure Baseline by simulating a cloud storage service without hiding data access patterns. Specifically, when a client accesses her data block in the server, the client directly sends the index of the data block to the server. After receiving the request, the server find the client's data block of interest according to the index and return it to the client. Note that the server uses data structure of array to store the data outsourced by client. The client will decrypt the encrypted block after receiving it and read/update the data block. Then the client encrypts the block and sends it back to the server. The server stores the updated block into its original position.

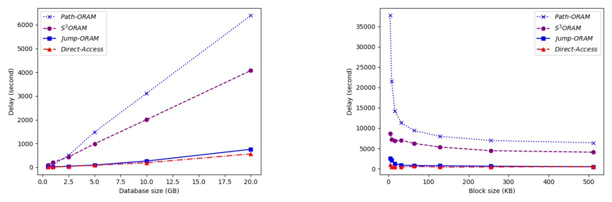
B. Implementation Configuration and Details

Software setting: We implement Jump-ORAM, Non-Secure Baseline, and Path-ORAM in C++. And, we use the external library libtomcrypt[1] to implement the symmetric AES-CTR cipher, an IND-CPA encryption. Note that the codes[2] to implement the S^3ORAM is in in [1]. Moreover, we have open-sourced the implementation of Jump-ORAM for public testing and updating.

Network and Hardware setting: To avoid the influence of an unstable network for experimental results, we conducted our experiments on one device: the PowerEdge T640 Tower Server equipped with Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz, RAM 29GB, and Ubuntu 20.04.4 LTS as an operating system. In other words, we locate the client and server on the same device when running the ORAM schemes, as mentioned earlier.

Database and block size: The database and block sizes range from 4 to 512KB and 0.5 to 20GB, respectively. To make the experimental results inherently comparable with previous work, we learned the indexes from [1].

C. Experimental Results



(a) Block size = 512KB (b) Database size = 20GB

Fig. 2. Jump-ORAM and its counterparts initialization time

Experimental results shows that Jump-ORAM's initialization efficiency performed better than Path-ORAM and S^3ORAM, and was almost identical to Non-Secure Baseline for all database sizes and block sizes. As shown in Figure 2, Jump-ORAM took around 758 seconds to initialize a large database (*e.g.*, 20GB) with 512-KB block, which is approximately 8.4× and 5.4× faster than Path-ORAM and S^3 ORAM, respectively, while being the same order of magnitude as Non-Secure Baseline.

Experimental results shows that the impact for the increase of initialization delay caused by increasing the database size for Jump-ORAM is less than its counterparts except for Non-Secure Baseline, which corresponds to the slope of Jump-ORAM is flatter than others. As shown in Figure 2a, the Jump-ORAM's time to initialize 10-GB and 20-GB database with 512-KB block are 264 and 758 seconds, respectively, but for Path-ORAM and S^3ORAM are 3120, 6391 seconds and 2009, 4072 seconds.

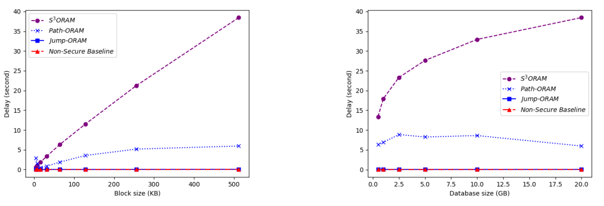
[1]https://github.com/libtom/libtomcrypt
[2]https://github.com/thanghoang/S3ORAM

Experimental results shows that for all ORAM schemes, the initialization time is negatively correlated with block sizes with fixed database size (*e.g.*, 20GB). As shown in Figure 2b, Jump-ORAM took around 637 and 505 seconds to create a 20-GB database with 256-KB and 512-KB block, respectively.

We scrutinized the total initialization delay of all ORAM schemes except for Non-Secure Baseline to investigate which components contributed the most to their initialization delay. Figure 4 shows detailed components of initialization delay for Jump-ORAM, S^3ORAM, and Path-ORAM varying the database sizes with a fixed 512-KB block. Note that for Path-ORAM, we equated the components of creating the database with dummy blocks and inserting real blocks into the database to the component of creating the database. In addition, for all ORAM schemes, the time to build metadata is negligible and, therefore, is hard to observe in Figure 4.

- Jump-ORAM: Experimental results show that the most significant delay in a database, the size of which is less than 10GB, was due to creating the database, which was approximately 88% of the overall delay when initializing with a 0.5-GB database from Figure 4a. On the contrary, when initializing the database, the size of which is greater or equal to 10GB, the contribution of creating the database is almost consistent with the delay in sending the database to the server. As shown in Figure 4a, the creating the database and sending the database to the server took about 364 and 393 seconds, respectively, with a 20-GB database.
- Path-ORAM: Experimental results show that the most of delay was due to creating database. As shown in Figure 4b, the delay of creating database contributed to about 68% of the overall delay. The time of sending the database to the server grows linearly concerning the database size. For example, the delay in sending the database to the server accounted for 48% of the overall delay with 20-GB database and 512-KB block.
- S^3ORAM: Experimental results show that the largest share of the total delay for all database sizes was the delay of creating the database, which accounted for approximately 67% of the overall delay Figure 4c. The delay of creating the database increases with increasing database sizes. For example, the delay of creating the database to initialize a 10-GB and 20-GB database took about 1352 and 2759 seconds, respectively.

Experimental results show that Jump-ORAM's access efficiency outperformed other ORAM schemes for all database and block sizes. As shown in Figure 3, Jump-ORAM took around 0.06 seconds to access the 512-KB block, which is approximately 100× and 633× faster than Path-ORAM and S^3ORAM, respectively with a 20-GB database. Moreover, the efficiency of Jump-ORAM is in the same order of magnitude as Non-Secure Baseline. For example, Jump-ORAM and Non-Secure Baseline took 0.06 and 0.02 seconds to access a 512-KB block.



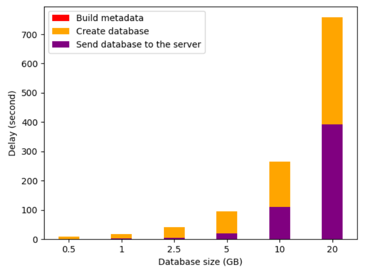
(a) Database size = 20GB (b) Block size = 512KB

Fig. 3. Access time of Jump-ORAM and its counterparts

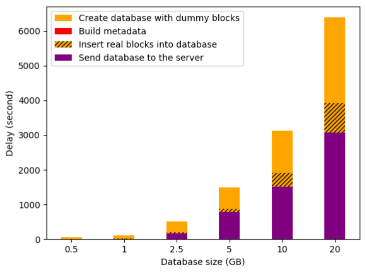
Experimental results show that although the delay of each ORAM scheme grows linearly with the block size, the slope of Jump-ORAM is much lower than that of its counterparts. Figure 3a illustrates the impact of increasing block size on the delay of Jump-ORAM and its counterparts. Given any block size range from 4 to 512KB, the Jump-ORAM's access delay fluctuated between 0.02 and 0.05 seconds. Figure 3b presents that the increasing database size hardly influenced the access delay for all schemes except for S^3ORAM. Since S^3ORAM depends on modular multiplication related to the entire database, its access delay is positively related to the database size.

Figure 5 presents the breakdown cost of access delay of Jump-ORAM and its ORAM counterparts. Note that since S^3RAM uses the SSS technique to achieve constant bandwidth overhead, we added server-computed metrics on top of the above elements to dissect its access latency. Next, we used the breakdown cost of access time in Figure 5 to analyze the factors that contributed the most to the total delay for each ORAM scheme.

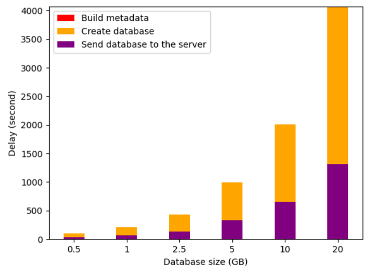
- Jump-ORAM: Experimental results show that the client-server communication and client computation dominated the total access delay. Note that encryption and decryption of data blocks are major components of client computation. As shown in Figure 5a, for all database sizes under the 512-KB block, client-server communication and client computation accounted for about 99% of the overall delay. Furthermore, the contributions to overall access delay from client-server communication and client computation are equally divided. For instance, the Jump-ORAM took 0.06 seconds to access a 512-KB block, where the costs of client-server communication and client computation are 0.021 and 0.034 seconds, respectively. Therefore, we can improve the performance of Jump-ORAM by reducing the costs of client-server communication and client computation.
- Path-ORAM: Experimental results show that the major factor influencing the access delay is the client



(a) Jump-ORAM with block size = 512KB

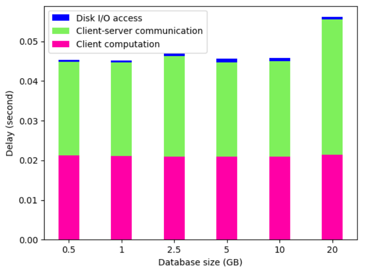


(b) Path-ORAM with block size = 512KB

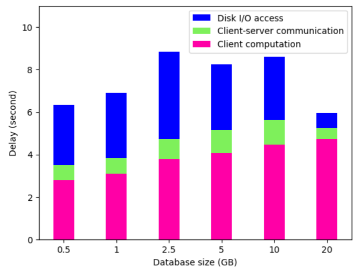


(c) S^3ORAM with block size = 512KB

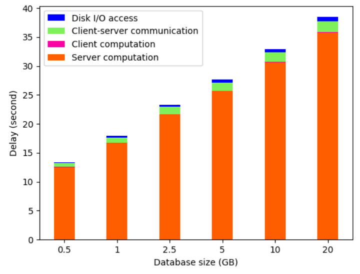
Fig. 4. Detailed breakdown cost of initialization time for ORAM schemes



(a) Jump-ORAM with block size = 512KB



(b) Path-ORAM with block size = 512KB



(c) S^3ORAM with block size = 512KB

Fig. 5. Detailed breakdown cost of access time for ORAM schemes

computation, which accounted for around 52% of the total delay for all database sizes except for 20GB under 512-KB block. Moreover, the second delay contributor is disk I/O access due to the $O(logN)$ times I/O access for every data access by the client. In addition, owing to the networking setting in this experiment, the transmission delay is negligible for each block transfer. Therefore, although Path-ORAM transferred $O(logN)$ blocks for each client's access, the client-server communication delay only accounted for 14% of the overall delay.

- S^3ORAM: Experimental results show that the ingredient of server computation contributed most to the S^3ORAM's overall delay. As shown in Figure 5c, for all database sizes under the 512-KB block, the server computation occupied approximately 92% of the overall access delay. Moreover, the client computation was negligible because the computation cost of recovery of the fetched block is small. The disk I/O access delay was also negligible since each server only read one block for every client's access.

VII. Conclusion

Although prior works (e.g.S^3ORAM) used the server's computation overhead to achieve $O(1)$ bandwidth overhead, their access delay suffers from the cost of server's computation overhead. In this paper, we proposed the Jump-ORAM that achieves the constant bandwidth overhead without suffering the cost of server-side computing overhead. The Jump-ORAM harnesses an iterative formula, array-based ORAM structure, and a secure rewriting strategy in an effective manner to achieve these objectives. We proved our scheme is statistically secure, and evaluated the Jump-ORAM and its counterparts in a simulated client-server model environment. Our experiments showed that the access efficiency of Jump-ORAM is two orders of magnitude faster than Path-ORAM and S^3ORAM, and is in same order of magnitude with the Non-Secure Baseline.

References

[1] T. Hoang, A. A. Yavuz, and J. Guajardo, "A Multi-server ORAM Framework with Constant Client Bandwidth Blowup," *ACM Transactions on Privacy and Security*, vol. 23, no. 1, pp. 1:1–1:35, 2020. [Online]. Available: https://doi.org/10.1145/3369108

[2] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 299–310. [Online]. Available: https://doi.org/10.1145/2508859.2516660

[3] B. Pinkas and T. Reinman, "Oblivious RAM Revisited," in *Advances in Cryptology – CRYPTO 2010*, ser. Lecture Notes in Computer Science, T. Rabin, Ed. Berlin, Heidelberg: Springer, 2010, pp. 502–519.

[4] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *in Network and Distributed System Security Symposium (NDSS*, San Diego, California, USA, 2012.

[5] E. Stefanov and E. Shi, "ObliviStore: High Performance Oblivious Cloud Storage," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 253–267, iSSN: 1081-6011.

[6] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, May 1996. [Online]. Available: https://doi.org/10.1145/233551.233553

[7] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with O((logN)3) Worst-Case Cost," in *Advances in Cryptology – ASIACRYPT 2011*, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds. Berlin, Heidelberg: Springer, 2011, pp. 197–214.

[8] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: 13th International Conference on Theory of Cryptography, TCC 2016," *Theory of Cryptography - 3th International Conference, TCC 2016-A, Proceedings*, pp. 145–174, 2016, publisher: Springer. [Online]. Available: http://www.scopus.com/inward/record.url?scp=84954156296&partnerID=8YFLogxK

[9] C. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov, "Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM," Tech. Rep. 1065, 2015. [Online]. Available: https://eprint.iacr.org/2015/1065

[10] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '12. USA: Society for Industrial and Applied Mathematics, 2012, pp. 157–167.

[11] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas, "Constants count: practical improvements to oblivious RAM," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. USA: USENIX Association, 2015, pp. 415–430.

[12] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Denver Colorado USA: ACM, Oct. 2015, pp. 837–849. [Online]. Available: https://dl.acm.org/doi/10.1145/2810103.2813649

[13] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 293–304. [Online]. Available: https://doi.org/10.1145/2382196.2382229

[14] S. Zahur, X. Wang, M. Raykova, A. Gascon, J. Doerner, D. Evans, and J. Katz, "Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation," in *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA: IEEE, May 2016, pp. 218–234. [Online]. Available: http://ieeexplore.ieee.org/document/7546504/

[15] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "A Retrospective on Path ORAM," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1572–1576, Aug. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8747366/

[16] M. Naveed, "The Fallacy of Composition of Oblivious RAM and Searchable Encryption," *IACR Cryptology ePrint Archive*, p. 668, 2015. [Online]. Available: https://eprint.iacr.org/2015/668

[17] E. Stefanov, C. Papamanthou, and E. Shi, "Practical Dynamic Searchable Encryption with Small Leakage," in *Proceedings 2014 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/programme/practical-dynamic-searchable-encryption-small-leakage/

[18] B. Zhao, Z. Chen, and H. Lin, "Cycle ORAM: A Practical Protection for Access Pattern in Untrusted Storage," *IEEE Access*, vol. 7, pp. 26 684–26 695, 2019, conference Name: IEEE Access.

[19] J. Dautrich, E. Stefanov, and E. Shi, "Burst ORAM: minimizing ORAM response times for bursty access patterns," in *Proceedings of the 23rd USENIX conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, pp. 749–764.

[20] P. Li and S. Guo, "Load balancing for privacy-preserving access to big data in cloud," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2014, pp. 524–528.

[21] "Usable PIR – NDSS Symposium." [Online]. Available: https://www.ndss-symposium.org/ndss2008/usable-pir/

[22] P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: practical access pattern privacy and correctness on untrusted storage," in *Proceedings of the 15th ACM conference on Computer and communications security*, ser. CCS '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 139–148. [Online]. Available: https://doi.org/10.1145/1455770.1455790

[23] T.-H. H. Chan, Y. Guo, W.-K. Lin, and E. Shi, "Oblivious Hashing Revisited, and Applications to Asymptotically Efficient ORAM and OPRAM," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, vol. 10624, pp. 660–690, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-70694-8_23

[24] M. T. Goodrich and M. Mitzenmacher, "Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, L. Aceto, M. Henzinger, and J. Sgall, Eds. Berlin, Heidelberg: Springer, 2011, pp. 576–587.

[25] K. G. Larsen and J. B. Nielsen, "Yes, There is an Oblivious RAM Lower Bound!" in *Advances in Cryptology – CRYPTO 2018*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds. Cham: Springer International Publishing, 2018, pp. 523–542.

[26] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 850–861. [Online]. Available: https://doi.org/10.1145/2810103.2813634

[27] E. Boyle and M. Naor, "Is There an Oblivious RAM Lower Bound?" in *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, ser. ITCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 357–368. [Online]. Available: https://doi.org/10.1145/2840728.2840761

[28] T.-H. Hubert Chan and E. Shi, "Circuit OPRAM: Unifying Statistically and Computationally Secure ORAMs and OPRAMs," in *Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2017, pp. 72–107. [Online]. Available: https://doi.org/10.1007/978-3-319-70503-3_3

[29] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing ORAM and Using It Efficiently for Secure Computation," in *Privacy Enhancing Technologies*, E. De Cristofaro and M. Wright, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–18.

[30] T. Mayberry, E.-O. Blass, and A. H. Chan, "Efficient Private File Retrieval by Combining ORAM and PIR," 2014. [Online]. Available: http://eprint.iacr.org/2013/086

[31] G. Liu, K. Li, Z. Xiao, and R. Wang, "Ps-oram: Efficient crash consistency support for oblivious ram on nvm," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 188–203. [Online]. Available: https://doi.org/10.1145/3470496.3527425

[32] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam, "Verifiable Oblivious Storage," in *Proceedings of the 17th International Conference on Public-Key Cryptography — PKC 2014 - Volume 8383*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 131–148. [Online]. Available: https://doi.org/10.1007/978-3-642-54631-0_8

[33] A. Anton, "Implementing onion oram: A constant bandwidth oram using ahe," 2016. [Online]. Available: https://github.com/aanastasov/onion-oram/blob/master/doc/report.pdf

[34] T. Moataz, E.-O. Blass, and T. Mayberry, "CHf-ORAM: A Constant Communication ORAM without Homomorphic Encryption," Tech. Rep. 1116, 2015. [Online]. Available: https://eprint.iacr.org/2015/1116

[35] T. Moataz, T. Mayberry, and E.-O. Blass, "Constant Communication ORAM with Small Blocksize," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 862–873. [Online]. Available: https://doi.org/10.1145/2810103.2813701

[36] A. Pujol and C. Thorpe, "Dog ORAM: A Distributed and Shared Oblivious RAM Model with Server Side Computation," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. Limassol, Cyprus: IEEE, Dec. 2015, pp. 624–629. [Online]. Available: https://ieeexplore.ieee.org/document/7431485/

[37] I. Abraham, C. W. Fletcher, K. Nayak, B. Pinkas, and L. Ren, "Asymptotically Tight Bounds for Composing ORAM with PIR," in *Public-Key Cryptography - PKC 2017 - 20th IACR International Conference on Practice and Theory in Public-Key Cryptography, Amsterdam, The Netherlands, March 28-31, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. Fehr, Ed., vol. 10174. Springer, 2017, pp. 91–120. [Online]. Available: https://doi.org/10.1007/978-3-662-54365-8_5

[38] E. Stefanov, E. Shi, and D. Song, "Towards Practical Oblivious RAM," Dec. 2012, number: arXiv:1106.3652 arXiv:1106.3652 [cs]. [Online]. Available: http://arxiv.org/abs/1106.3652