Constant Communication Oblivious RAM

Tarik Moataz
Colorado State University
IMT Telecom
tmoataz@cs.colostate.edu

Travis Mayberry
United States Naval Academy
travism@ccs.neu.edu

Erik-Oliver Blass
Airbus Group Innovations
erik-oliver.blass@airbus.com

ABSTRACT

There have been several attempts recently at using homomorphic encryption to increase the efficiency of Oblivious RAM protocols. One of the most successful has been Onion ORAM, which achieves $O(1)$ communication overhead with polylogarithmic server computation. However, it has a number of drawbacks. It requires a very large block size of $B = \Omega(\log^5 N)$, with large constants. Although it needs only polylogarithmic computation complexity, that computation consists mostly of expensive homomorphic multiplications. Finally, it achieves $O(1)$ communication complexity but only amortized over a number of accesses. In this work we aim to address these problems, reducing the required block size to $\Omega(\log^3 N)$, removing almost all of the homomorphic multiplications and achieving $O(1)$ worst-case communication complexity. We achieve this by replacing their homomorphic eviction routine with a much less expensive permute-and-merge one which eliminates homomorphic multiplications while maintaining the same level of security. In turn, this removes the need for layered encryption that Onion ORAM relies on and reduces both the minimum block size and worst-case bandwidth.

1. INTRODUCTION

With cloud storage is becoming increasingly popular and relied upon by both enterprise and individual users, ensuring proper security and privacy is a critical research problem. Reports indicate that up to 88% of organizations [19] are using public cloud infrastructure for at least some of their data. It is fairly straightforward to encrypt that data, but that is not always enough. Where, when and how often a user accesses their data can reveal as much about it as the plaintext itself. We call this information a user's *access pattern*. For instance, observing that an investment bank has repeatedly accessed their files on a specific company may reveal that they plan to invest in that company. Crucially, there is no easy way to bound what information you might leak as part of your access pattern, especially when an adversary can correlate those accesses with other outside (potentially public) information he might have.

Oblivious RAM is a tool that was designed to solve exactly this problem. Given a set of accesses to a block storage device, an ORAM algorithm allows a user to perform them on an untrusted storage device in such a way that an adversary observing those accesses cannot determine which block the user was reading/writing. This generally involves shuffling and reencrypting the data somehow each time it is read or written to in order to unlink two accesses to the same block. Unfortunately, ORAM has traditionally been very expensive to implement, causing over a thousand-fold increase in communication over unprotected accesses.

Recently there has been a flurry of research on ORAM that has managed to drastically decrease communication overhead with a new tree-based paradigm [20]. Building on that, Stefanov et al. [22] introduced Path ORAM which, along with some derivative works, is the most efficient construction currently known. However, it still requires polylogarithmic communication overhead which can result in over a hundred-fold slowdown and may not be usable for many cloud applications given that cost is a driving factor in outsourcing data. Along with work on pure Oblivious RAM, Mayberry et al. [15] introduced the idea that communication overhead can be greatly reduced if the storage device is also considered to have some computational ability, which it generally does in a cloud setting. Using recent advances in homomorphic encryption, a small amount of computation on the server can be leveraged to cut a significant amount of communication to the client.

Furthering this research, Devadas et al. [4] have recently proposed a hybrid ORAM-with-computation scheme that achieves $O(1)$ communication overhead. They achieve this by consecutively wrapping blocks in further layers of encryption as they proceed down the tree, effectively forming an "onion" out of the blocks. Unfortunately, it still has some major drawbacks:

1. The $O(1)$ overhead is only amortized. Worst case complexity is still $O(\log N)$, with relatively large constants.
2. Their scheme requires that the block has a very large size, $\Omega(\log^5 N)$. In practice, it can be up to 30 MB for reasonably sized databases.
3. The onion part of their scheme requires a large number of homomorphic multiplications, which are computationally very expensive. Depending on the encryption scheme used, overhead on the server may outweigh any communication saved.

In this work we aim to tackle all these problems. We start by showing that the homomorphic multiplications, and in fact the nesting "onion" nature of their solution, is not necessary. With careful application of an oblivious merging algorithm, all movement of blocks through the tree can be done with only homomorphic addition, resulting in a more computationally efficient algorithm. This also reduced the required block size by a $O(\log^2 N)$ factor and, we will show, allows for $O(1)$ communication in the worst case as well as amortized. Finally, we show via experimental evaluation that our scheme requires only a small storage overhead compared to Onion ORAM and that, for practical parameter values, we achieve significant improvement in block size and number of homomorphic operations. Figure 1 summarizes our improvements when compared to Onion ORAM.

2. BACKGROUND: ONION ORAM

We start by briefly introducing the main idea of Onion ORAM [4]. We then analyze its complexity to motivate our improvements.

2.1 Overview

An Oblivious RAM is a block-based storage protocol whereby a user can outsource some data to an untrusted server, and that server does not learn anything about the pattern of accesses that the user performs on that data. For instance, whether the user accesses the same block many times in a row, or each block individually in sequence, the server will not be able to distinguish between these two access patterns. In fact, a secure ORAM guarantees that *any* two access patterns will be indistinguishable from the perspective of the server. This is accomplished by periodically moving, shuffling and reencrypting the data so that correlations between accesses are lost. A twist on that model introduced by Mayberry et al. [15], and used in Onion ORAM, is that instead of the traditional ORAM server definition where it only stores the data passively, Onion ORAM assumes that the server can also perform computations.

Onion ORAM is a tree-based ORAM, and shares many qualities with existing schemes [5, 20, 22]. Most importantly, data blocks are stored in a tree where each node of the tree is a "bucket" which contains some number of blocks. When blocks are added to the ORAM, they start at the root of the tree and are tagged as belonging to one of the leaf nodes. As the lifecycle of the ORAM continues, blocks percolate from the root to their assigned leaf node through a process called *eviction*. This way, a block can be located at any time by reading the path from its target leaf back to the root, since it is guaranteed to always reside on this path. The eviction process maintains a proper flow of blocks from the root to the leaves so that no buckets overflow with too many blocks. This is usually accomplished by picking a path in the tree, from root to a particular leaf node, and pushing all the blocks on that path as far as possible down the path toward the leaf node.

The contribution of Onion ORAM is then that it achieves constant communication complexity in the number of ORAM elements N, while only requiring polylogarithmic computation on the server. Although the client exchanges many pieces of data back and forth with the server, the key to having $O(1)$ communication complexity is that the size of one data block, B, dominates the communication. All other messages, ciphertexts, etc. are collectively small compared to the actual data being retrieved. Therefore it might be more intuitive to say that communication is $O(B)$. However, it is customary in ORAM literature to refer to the communication complexity in terms of multiplicative overhead, i.e. the cost compared to retrieving the same data without any security. Everything is then divided by B and we get to the notion of $O(1)$ communication. Note that $O(1)$ communication is not difficult if you allow unrestricted computation (FHE for instance achieves this trivially), so the limit to polylogarithmic computation is important.

The main idea behind Onion ORAM is an oblivious shuffling based on (computational) Private Information Retrieval (PIR). Therewith, ORAM read, write, and eviction operations can be performed without the client actually downloading data blocks and doing the merging themselves. This saves a huge amount of communication when compared to existing schemes like Path ORAM. Compared to existing tree-based ORAM schemes, Onion ORAM introduces a *triple eviction* that empties all buckets along the path instead of only pushing some elements down and leaving others at intermediate points in the tree. Elements in any evicted bucket will be pushed towards both children, thereby ensuring that after an eviction the entire evicted path is empty aside from the leaves. The authors take advantage of the fact that if you choose which path to evict by *reverse lexicographic ordering*, then you are always guaranteed during an eviction that the sibling of every node on your path will already be empty from a previous eviction. This allows for the entire process to be done efficiently and smoothly, because the entire contents of a parent can be copied into the empty bucket and there is no need to worry about overwriting what was there because it is necessarily empty.

This triple eviction is accomplished by sending a logarithmic number of oblivious shuffling vectors to the server. These vectors, encrypted with an additively homomorphic encryption, obliviously map an old block of the parent bucket to a new position in the child. This operation is made by a matrix multiplication between the vector sent to the server and the bucket. Considering the size of the bucket as logarithmic, this algebraic computation should be performed a polylogarithmic number of times. This results that each block is encrypted, without transitional decryption, a logarithmic number of times, hence, the attributed name "onion".

The above results in an *amortized* ORAM with constant communication complexity and constant client-memory, in the number of elements N stored in the ORAM. If λ refers to the security parameter, to obliviously retrieve a block of size B bits from the server, Onion ORAM requires $O_\lambda(B)$ bits communication, where $O_\lambda(.)$ is a notation to denote the amortized asymptotics over $\lambda \in \omega(\log N)$. This is an amortized improvement over related work such as Shi et al. [20]'s ORAM with $O(B\log^3 N)$ communication complexity or Path ORAM [22] with either $O(B\log^2 N)$ or $O(B\log N)$ complexity (depending on block size B).

2.2 Analysis

As noted above, $O(1)$ communication does not imply that blocks are the only exchanged information between client and server. In Onion ORAM, the client still needs to retrieve meta-information and send PIR vectors for PIR reads and PIR writes. Thus, Onion ORAM chooses the block size such that all communication between server and client is asymptotically dominated by block size B. That is, if $B \in O(|\text{meta-information}| + |\text{PIR vectors}|)$, then Onion ORAM has constant communication complexity.

2.2.1 Large Block Size

Consequently, to achieve constant communication complexity, Onion ORAM requires a large block size B. For a security parameter γ in the order of 2048 Bytes, bucket size $z = \Theta(\lambda)$, and number of elements N, the block size B in Onion ORAM is in $\Omega(\gamma\lambda\log^2 N)$. This is a significant increase over $B \in \Omega(\log N)$ required by related work [20, 22]. Generally, requiring a large block size renders ORAMs impractical for many real world scenarios where the block size is fixed and simply predetermined by the application. To mitigate the problem, Onion ORAM uses Lipmaa's PIR [14] instead of an straightforward additively homomorphic PIR [12]. This decreases block size to $B \in \Omega(\gamma\log^2\lambda\log^2 N)$. Factor λ is replaced by $\log^2\lambda$. On a side note, observe that using Lipmaa's PIR might not result in much (or any) gain in practice. Parameter λ is a security parameter such that $\lambda \in \omega(\log N)$, typically small, and therefore "close" to $\log^2\lambda$. For example, for $\lambda = 80$, $\log^2\lambda = 40$ is in the same order of magnitude. Since Lipmaa's method requires substantially more computation than the straight-

Scheme	Block size	Simplified block size	Worst-case bandwidth	# additions	# multiplications
Onion ORAM	$\Omega(\gamma\lambda \log^2 N)$	$\Omega(\log^5 N)$	$O(\lambda)$	$\Theta(\lambda^2 \log^2 N)$	$\Theta(\lambda^2 \log^2 N)$
C-ORAM	$\Omega(\lambda[\log\lambda \log N + \gamma])$	$\Omega(\log^3 N)$	$O(1)$	$\Theta(\lambda \log N)$	$\Theta(\lambda)$

Figure 1: Comparison of Onion ORAM and C-ORAM, containing block size, worst-case bandwidth, and number of homomorphic additions and multiplications. Simplified block value is a looser bound which allows for easier comparison and relies on the fact that $\lambda = \omega(\log N)$ and $\gamma = O(\lambda^2)$.

forward approach, the small communication gain is likely to be outweighed by additional computation time.

Onion ORAM block size example: For security parameter $\gamma = 2048$, number of elements $N = 2^{20}$, and security parameter $\lambda = 80$, the block size must be at least $B = 2048 \cdot \log^2(80) \cdot 20^2 \approx 33$ MBits. Thus, the dataset size equals $2^{20} \cdot 33 \cdot 10^6 \approx 35$ TBits. This computation is very rough and does not take into account additional, hidden constants such as the constant for the additively homomorphic cipher chunk in $\Omega(\gamma \log N)$, or smaller, yet still significant constants, like the fact that downloads have corresponding uploads which multiplies everything by 2.

Requiring blocks of size at least 4 MBytes to store $N = 2^{20}$ elements is impractical for many real world applications. In conclusion, Onion ORAM can only be applied to very special data sets with very large block sizes.

2.2.2 Amortized Complexity

Besides the large block sizes, a second problem with Onion ORAM is that communication complexity is constant only in an *amortized* analysis. PIR write operations in Onion ORAM involve additively homomorphic encryption that add encryption layers during eviction. Each layer increases the size of the blocks. Consequently, after every $\chi = \Theta(\lambda)$ read or write operations, the client must download the leaf of an evicted path and peel off encryption layers. In addition, the eviction based on PIR writes is λ times more expensive than ORAM read and write operations that are only based on PIR read. So, the amortized eviction every χ operations reduces the additional cost of the eviction itself and the encryption layer peeling. In the worst case, the eviction triggered after every χ ORAM operations results in a communication complexity of $O(\lambda B)$ with $\lambda \in \omega(\log N)$.

To sum up, the main downsides of Onion ORAM are: (1) a large block size rendering Onion ORAM impractical for many real world applications, (2) amortized costs that hide a worst case factor of z.

3. CONSTANT COMMUNICATION ORAM

Overview: To achieve our increased efficiency and lower block size, we present a novel, efficient oblivious bucket merging technique for Onion ORAM that replaces its expensive layered encryption. We apply our bucket merging during ORAM eviction. The content of a parent node/bucket and its child node/bucket can be merged obliviously, i.e., the server does not learn any information about the load of each bucket. The idea is that the client sends a permutation Π to the server. Using this permutation, the server aligns the individual encrypted blocks of the two buckets and merges them into a destination bucket. The client chooses the permutation such that blocks containing real data in one bucket are always aligned to empty blocks in the other bucket. As each block is encrypted with additively homomorphic encryption, merging two blocks is a simple addition of ciphertexts. For the server, merging is oblivious, because, informally, any permutation Π from the client is indistinguishable from a randomly chosen permutation.

For buckets of size $O(z)$, our oblivious merging evicts elements from a parent bucket to its child with $O(z \log z)$ bits of communication instead of $O(\gamma z^2)$ of Onion ORAM. As a result of applying our merging technique, we only need a *constant* number of PIR reads and writes for ORAM operations.

Based on our merging technique, we now present increasingly sophisticated modifications to Onion ORAM to reduce its costs. We call the resulting ORAM, i.e., Onion ORAM with our modification, C-ORAM. As a warm up, we present a technique allowing amortized constant communication complexity with a smaller block size B in $\Omega(z \log z \log N + \gamma z \log N)$. Our second and main technique improves the first one and results in constant worst case communication complexity. The block size of this second construction is in $\Omega(z \log z \log N + \gamma z)$.

3.1 Oblivious Merging

Oblivious merging is a technique that obliviously lines up two buckets in a specific order and merges them into one bucket. Using this technique, we can evict real data elements from a bucket to another by permuting the order of blocks of one of them and then adding additively homomorphically encrypted blocks. Oblivious merging is based on an oblivious permutation generation that takes as input the *configurations* of two buckets and outputs a permutation Π. A configuration of a bucket specifies which of the blocks in the bucket are real blocks and which are empty. Permutation Π arranges blocks in such a way that there are no real data elements at the same position in the two blocks.

3.1.1 C-ORAM Construction

C-ORAM keeps Onion ORAM's main construction. That is, C-ORAM is a tree-based ORAM composed of a main tree ORAM storing the actual data and a recursive ORAM storing the position map. The position map consists of a number of ORAM trees with linearly increasing height mapping a given address to a tag. For n elements stored in the ORAM, the communication needed to access the position map is in $O(\log^2 N)$. As with all recent tree-based ORAMS, the recursive position map's communication complexity is dominated by the block size. For the remainder of this paper, we therefore restrict our description only on C-ORAM's main data tree.

Let N be a power of 2. C-ORAM is a binary tree with L levels and 2^L leave nodes. Each node/bucket contains $\mu \cdot z$ blocks. Here, z is the number of slots needed to hold blocks as in Onion ORAM and μ is a multiplicative constant that gives extra room in the buckets for noisy blocks, a detail we will cover below which is important for our construction. We maintain the same relation between N, L and z as in Onion ORAM, namely $N \leq z \cdot 2^{L-1}$. This ensures only constant storage overhead of about 4μ. Note that $L = \Theta(\log n)$.

Each block in a C-ORAM bucket is encrypted using an additively homomorphic encryption, e.g., Pailler's or Damgard-Jurik's cryptosystem. Also, each bucket contains IND-CPA encrypted meta-information, *headers*, containing additional information about a bucket's contents.

3.1.2 Headers

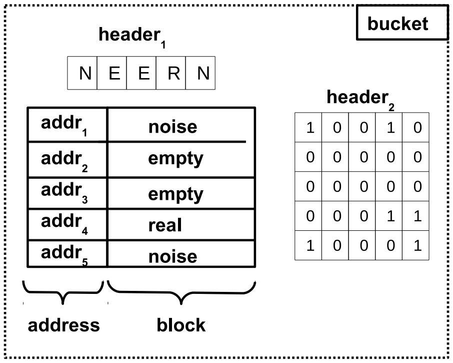


Figure 2: C-ORAM bucket structure

Bucket headers are an important component in C-ORAM as they determine how oblivious permutations are generated. A bucket header is comprised of two parts: the first part stores for each block whether it is noisy, contains real data or is empty. The second part stores the block tags. More formally, the header is composed of two vectors header$_1$ and header$_2$. Vector header$_1$ has length $\mu \cdot z$, and each element is either noisy, empty or real. Thus, each element has a size of two bits. The total size of this vector is in $O(\mu z)$. header$_2$ is a $(\mu \cdot z \times \log N)$ binary matrix. The rows represent the address of the blocks. Finally, as with all tree based ORAMS, each block in a bucket also contains the encryption of its address. That is, the address of each block is encrypted separately from the block itself. We show a high level view of a C-ORAM bucket in Fig 2.

3.2 C-ORAM: First Construction

To prepare for our main contribution, we start by presenting a new technique allowing amortized constant communication complexity with a smaller block size.

3.2.1 Overview

To access an element in C-ORAM, i.e., read or write, the client first fetches the corresponding tag from the position map. This tag defines a unique path starting from the root of the ORAM tree and going to a specific leaf given by the tag. The element might reside in any bucket on this path. To find this element, we make use of a PIR read [12] that will be applied to each bucket. To verify whether the block exists in a bucket, the client downloads the encrypted headers of each bucket. Therewith, the client can generate a PIR read vector retrieving the block from a bucket. To preserve the scheme's obliviousness, the client sends PIR read vectors for each bucket on the path. Once the block has been retrieved, the client can modify the block's content if required, then insert it back into the root of the C-ORAM tree using PIR write. This is the standard Path-PIR behavior to read from or write into blocks [15].

Eviction in our first construction takes place after every $\chi = O(z)$ access operations. As in Onion ORAM, a path in C-ORAM is selected following deterministic reverse lexicographic order. Then, the entire root of the ORAM tree is downloaded, randomly shuffled and written back (additively homomorphically) encrypted. Finally, the eviction is performed by repeatedly applying an oblivious merge on buckets along the selected path. Any bucket belonging to this path is obliviously merged with its parent while the other child of the parent will be overwritten by a copy of the parent bucket. We call the former bucket on the path the *destination* bucket and the latter one its *sibling* bucket.

Before starting the eviction of a specific path, an invariant of the eviction process is that siblings of buckets of this path are empty, except the leaves. After the eviction, all buckets belonging to the evicted path will be empty except the leaf [4]. Note that siblings of this path, after the eviction, will not be empty anymore. See Fig 3 for a sample eviction with $N = 8$.

Sibling buckets, since their are simply copies of their parents, will contain blocks with tags outside the subtree of this bucket. These blocks are called *noisy* blocks as they do not belong into this subtree and are essentially leftover "junk". Now for correctness, in our construction, we will guarantee that the number of noisy blocks in any bucket is upper bounded. So, there will always be space for real elements in a bucket and will not overflow.

Elements in each bucket are encrypted using additively homomorphic encryption, respectively. Given two buckets B_1 and B_2, oblivious merging will permute the position of blocks in B_1 such that there are no real or noisy element at the same positions in B_1 and B_2. Consequently, if there is a real element in the i^{th} position in B_1, then for the scheme to be correct, the i^{th} position in B_2 should be empty. The following addition of elements at the same position in B_1 and B_2 will preserve the value of the real element. After χ operations, we also download the leaf bucket to delete its noisy blocks.

3.2.2 Details and Analysis

Let $\mathcal{P}(tag)$ denote the path starting from the root and going to the leaf identified by tag. The path is composed of $\log N + 1$ buckets including the root. $\mathcal{P}(tag, i)$ refers to the bucket at the i^{th} level of $\mathcal{P}(tag)$. For example, $\mathcal{P}(tag, 0)$ is the root bucket. $\mathcal{P}_s(tag, i)$ is the sibling of bucket $\mathcal{P}(tag, i)$. We denote by $[N]$ the set of integers $\{1, \cdots, N\}$, $x \xleftarrow{\$} [N]$ uniformly sampling a random element from set $[N]$, and by χ the period of eviction which is in $O(z)$. Identity stands for an empty bucket containing only encryptions of zero.

Algorithm 1 presents details of the access operation. An access can be either an ORAM Read or a Write operation. The only difference between the two is that a write changes the value of the block before putting it back in the root. The access operation invokes a PIR read algorithm, see Algorithm 2 that obliviously retrieves a block. Algorithm 3 shows the eviction where elements percolate towards their leafs using oblivious permutations, see Algorithm 4.

Block size asymptotics: The following asymptotical analysis will be in function of z, N, and γ. z is the size of the bucket, N the number of elements, and γ the length of the ciphertext of the additively homomorphic encryption. The communication complexity induced by an ORAM access operation comprises a PIR read operation and the eviction process (happening every $\chi \in O(z)$ accesses). The size of the bucket is $\mu \cdot z$, but we will show in our security analysis section later that μ is a constant. Therefore, we ignore it in our analysis.

First, the client performs PIR reads $L + 1$ times. For this, the client has to download all addresses in the path, i.e., $O(zL \log N)$ bits. Also, the client should send a logarithmic number of PIR read vectors $\mathcal{V}$ with size $O(\gamma z L)$ bits. Note that the computation of PIR read vectors outputs for all but one buckets blocks of encrypted zeros. Instead of sending back a logarithmic number of blocks to the client, the server only sends a single block, the summation of all the blocks output, cf. Algorithm 1. Thus, the client only retrieves a single block B. A PIR read applied to all buckets of the path induces an overhead in $O(zL \log N + \gamma z L + B)$.

For the eviction, the client downloads header$_1$ and the i^{th} column of header$_2$ and sends permutations for all buckets in the path. Thus, the overhead induced by the permutations is $O(Lz \log z)$

bits. Also, after every $\chi = O(z)$ operations, the client downloads the root and one leaf, which has $O(zB)$ communication complexity. Amortized, for each operation we have $O_z(B)$ communication complexity (amortized over z).

In conclusion, each access has $O_z(zL\log N + \gamma zL + z\log(z)L + B)$ communication complexity. To have a constant communication communication complexity in B, the block size should be

$$\begin{aligned} B &\in \Omega(z\,L\log N + \gamma zL + Lz\log z) \\ &\in \Omega(\lambda\log^2 N + \gamma\lambda\log N). \end{aligned} \quad (1)$$

Expression 1 is a consequence of $z = \Theta(\lambda)$, $\lambda \in \omega(\log n)$ and $L \in \Theta(\log N)$. In general, $\gamma = O(\lambda^2)$, which means that the value $\lambda\cdot\log^2 N$ is negligible against $\gamma\cdot\lambda\cdot\log N$, then $B \in \Omega(\gamma\lambda\log N)$.

The block size of our first modification is already a $\log N$ multiplicative factor improvement over the block size of Onion ORAM. However, in practice, this value is still large. The main idea in our second construction is based on the following observation. The block size has exactly the same asymptotic as transmitted vectors $\mathcal{V}$. So to improve the block size, we change the way we are accessing the ORAM.

```
Input: Operation op, address adr, data data, counter ctr, state st
Output: Block $B$ associated to address addr
// Fetch tag value from position map
tag ← posMap(adr);
posMap(adr) $\xleftarrow{\$}$ $[N]$;
if $\mathsf{ctr} = 0 \mod (\chi)$ then
    Download root bucket, refresh encryptions, randomize order of real elements;
    Evict(st);
else
    for $i$ from 0 to $L$ do  $B = B +$ PIR-Read(adr, $\mathcal{P}$(tag, $i$)) ;
end
if op = write then  set $B$ = data ;
ctr = ctr + 1;
Upload IND-CPA encrypted block to root $\mathcal{P}$(tag, 0);
```

Algorithm 1: Access(op, adr, data, ctr, st): C-ORAM access operation, 1$^{\text{st}}$ construction

```
Input: Bucket $\mathcal{P}$(tag, level)), address adr
Output: Block $B$
Retrieve and decrypt addresses Addr of bucket $\mathcal{P}$(tag, level));
// Compute the PIR-Read vector $\mathcal{V}$ in client side
if adr ∈ Addr then
    // Retrieve the index $\alpha$
    $\alpha$ = Addr[addr];
    for $i$ from 1 to $\mu \cdot z$ do
        if $i \neq \alpha$ then  $\mathcal{V}_i \xleftarrow{\$}$ ENC(0)  else
        $\mathcal{V}_i \xleftarrow{\$}$ ENC(1) ;
    end
else
    for $i$ from 1 to $\mu \cdot z$ do  $\mathcal{V}_i \xleftarrow{\$}$ ENC(0) ;
end
// Retrieve block in server side
Parse bucket $\mathcal{P}$(tag, level) as $(\mu \cdot z \times |B|)$ binary matrix $\mathcal{M}$;
$B = (\sum_{i=1}^{\mu\cdot z} \mathcal{V}_i \cdot \mathcal{M}_{1,i}, \cdots, \sum_{i=1}^{\mu\cdot z} \mathcal{V}_i \cdot \mathcal{M}_{|B|,i})$;
```

Algorithm 2: PIR-Read(adr, $\mathcal{P}$(tag, level))

3.3 C-ORAM: Second Construction

We start by further reducing the block size – again by a multiplicative factor of $\log N$ compared to our first construction. Then,

```
Input: State st
Output: Evicted path and updated state st
for $i$ from 0 to $\log n - 1$ do
    Retrieve header$_1^i$ and header$_1^{i+1}$;
    Retrieve C$_i$ and C$_{i+1}$ respectively the $i^{th}$ and the $(i+1)^{\text{th}}$
    column of header$_2^i$ and header$_2^{i+1}$ of the bucket $\mathcal{P}$(st, $i$) and
    $\mathcal{P}$(st, $i+1$);
    $\pi \xleftarrow{\$}$ GenPerm((header$_1^i$, C$_i$), (header$_1^{i+1}$, C$_{i+1}$)), generate
    the oblivious permutation $\pi$;
    // Merge the parent and destination bucket
    $\mathcal{P}(\mathsf{st}, i+1) = \pi(\mathcal{P}(\mathsf{st}, i)) + \mathcal{P}(\mathsf{st}, i+1)$;
    if $i < L - 1$ then
        // Copy the parent bucket into its
           sibling
        $\mathcal{P}_s(\mathsf{st}, i) = \mathcal{P}(\mathsf{st}, i)$;
    else
        // Merge the last bucket with the sibling
           leaf
        Retrieve header$_1^{i+1}$ and C$_{i+1}$ from the sibling leaf;
        $\pi \xleftarrow{\$}$ GenPerm((header$_1^i$, C$_i$), (header$_1{}^{i+1}$, C$_{i+1}$));
        $\mathcal{P}(\mathsf{st}, i+1) = \pi(\mathcal{P}(\mathsf{st}, i)) + \mathcal{P}(\mathsf{st}, i+1)$;
    end
    Update(header$_1^i$) and store it with bucket $\mathcal{P}_s$(st, $i$);
    Update(header$_1^{i+1}$) and store it with bucket $\mathcal{P}$(st, $i+1$);
    $\mathcal{P}$(st, $i$) = Identity;
end
```

Algorithm 3: Evict(st), eviction process

```
Input: Configuration of buckets $A$ and $B$
Output: A permutation randomly lining up bucket $B$ to bucket $A$
// Slots in $A$ and $B$ start either empty, full
   or noisy; mark slots in $A$ as assigned if
   block from $B$ is assigned in $\pi$
Let $x_1, x_2$ be the number of empty and noisy slots in $A$;
Let $y_1, y_2$ be the number of full and noisy slots in $B$;
$d_1 = x_1 - y_1$;
$d_2 = x_2 - y_2$;
for $i$ from 1 to $k$ do
    case $B[i]$ is full  $z \xleftarrow{\$}$ all empty slots in $A$ ;
    case $B[i]$ is noisy
        if $d_2 > 0$ then
            $z \xleftarrow{\$}$ all noisy slots in $A$;
            $d_2 = d_2 - 1$;
        else
            $z \xleftarrow{\$}$ all empty slots in $A$;
        end
    end
    case $B[i]$ is empty
        if $d_1 > 0$ then
            $z \xleftarrow{\$}$ all non-assigned slots in $A$;
            $d_1 = d_1 - 1$;
        else
            $z \xleftarrow{\$}$ all full slots in $A$;
        end
    end
    $\pi[i] = z$;
    $A[z]$ = assigned;
end
return $\pi$;
```

Algorithm 4: GenPerm(A, B), oblivious permutation generation

we improve asymptotics even for the worst case. Recall that in our first construction, the worst case involves a blow-up of $O(z)$, because during eviction the client needs to download $O(zB)$ bits. In our second and main construction, the eviction remains exactly the same, and our focus will only be on the ORAM access.

3.3.1 Overview

In our first modification, we perform a PIR read per bucket during an access. Contrary, we now perform an oblivious merge to find out the block to retrieve. For an ORAM access to tag, our idea is to perform a special evict of path $\mathcal{P}(tag)$. We push all *real* elements in $\mathcal{P}(tag)$ towards the leaf and then simply access the leaf bucket. So, we preserve access obliviousness and make sure that the element we want is pushed into leaf bucket tag.

This approach comes with several challenges. We must preserve the bucket distribution, i.e., we have to maintain sibling emptiness property, as guaranteed by the reverse lexicographic eviction, before evicting any path. Instead of deterministically selecting a path for eviction, we choose paths randomly. However, using randomized eviction, we still have to guarantee empty siblings on the evicted path. By randomly evicting a path, we might copy a bucket in its sibling which is not empty resulting therefore in a correctness flaw.

Our approach will be to temporarily clone the path $\mathcal{P}(tag)$. The clone of $\mathcal{P}(tag)$ serves to simulate the eviction towards the leaf bucket, and we remove the clone after the access operation. We apply the oblivious merging on the bucket of this cloned path, and at the end we will have all real elements in the leaf bucket of the cloned path. Finally, we apply a PIR read to retrieve the block.

Besides, to get rid of the amortized cost and have a scheme that only requires a constant bandwidth in the worst case, we make use of a PIR write operation that will be performed during every access. In the first construction, we have to shuffle the root bucket since oblivious merging has to be performed on random buckets for security purposes. Moreover, we need to eliminate noisy blocks from the leaf buckets and therefore after each χ operations, the client downloads the evicted leaf to eliminate all noisy blocks. In C-ORAM second construction, we are evicting after every access, then we are certain that the root bucket is always empty after an eviction. The first PIR write operation that we perform will randomly insert the block in an empty root bucket after any access obliviously. The second use of PIR write is to delete the retrieved element from the leaf. In fact, we can also delete noisy blocks by the same tool but a PIR read is needed to retrieve first the noisy block that we will overwrite with a PIR write. We dedicate Section 4.2 to analyze security and correctness of our modification.

3.3.2 Details and Analysis

Algorithm 5 presents the core of our second construction. Now, instead of performing a logarithmic number of PIR reads, we only invoke an Evict-Clone to read a block, cf. Algorithm 6. Evict-Clone uses our oblivious merging, together with one PIR read to retrieve a block. Moreover, we evict after every access. In order to eliminate noisy blocks that have been percolated to the leaf bucket, we use a PIR write to delete the noisy block, cf. Algorithm 7.

Block size asymptotics: The access operation in C-ORAM comprises eviction, one eviction in the cloned path, a PIR read, and two PIR writes. The size of the headers are negligible compared to the PIR read and write vectors. Thus, we avoid including them in our asymptotics' details for sake of clarity.

First, the eviction always involves an overhead of $O(zL \log z)$. Evict-Clone performs one PIR read in addition to the regular evict. Finally, we retrieve the block of size B. Therefore, the overhead induced by these steps is $O(zL \log z + z \log N + \gamma z + B)$.

Adding the two PIR write and single PIR read operation will not change asymptotical behavior since the number of these operations is constant in N.

In conclusion, to have a bandwidth that is constant in block size B, the block size should be $B \in \Omega(zL \log z + z\gamma)$.

Having $z \in \Theta(\lambda)$, $\lambda \in \omega(\log N)$ and $L \in \Theta(\log N)$, $B \in \Omega(\lambda[\log N \log \lambda + \gamma])$.

In practice, $\gamma \in O(\lambda^2)$ so γ dominates $\log N \log \lambda$. Therefore, block size B is $B \in \Omega(\gamma\lambda)$.

Our second construction results in a worst case cost lower than the amortized cost of our first construction, but also omits inefficient PIR reads performed for ORAM access. This second construction improves the blocks size by a multiplicative factor of $\log^2 N$ compared to Onion ORAM in the worst case.

As you can see, the main overhead of C-ORAM block size comes from the size of the ciphertext γ. Recall that $\gamma \in O(\lambda^2)$, therefore the smaller the additively homomorphic ciphertext will get, the smaller the block size of C-ORAM will be.

```
Input: Operation op, address adr, data data, state st
Output: Block $B$ associated to address adr
// Fetch tag value from position map
tag $\leftarrow$ posMap(adr);
posMap(adr) $\xleftarrow{\$}$ $[N]$;
// Retrieve desired block
$B$ =Evict-Clone(adr, tag);
if op = write then set $B$ = data ;
// Select a random position in the root bucket
$\mathsf{pos}_1 \xleftarrow{\$} [\mu \cdot z]$;
// Write back the block to the empty root
PIR-Write($\mathsf{pos}_1, B, \mathcal{P}(\mathsf{st}, 0)$);
Evict(st);
// Select a random noisy block position from
   the header of the leaf $\mathcal{P}(\mathsf{st}, L)$
$\mathsf{pos}_2 \xleftarrow{\$} \mathsf{header}^L$;
$N$ =PIR-Read($\mathsf{pos}_2, \mathcal{P}(\mathsf{st}, L)$);
// Write back the negation of the noisy block
PIR-Write($\mathsf{pos}_2, -N, \mathcal{P}(\mathsf{st}, L)$);
```

Algorithm 5: Access(op, adr, data, st): C-ORAM access operation, 2^{th} construction

```
Input: Leaf tag and address adr
Output: Block $B$
Create a copy of the C-ORAM path $\mathcal{P}(\mathsf{tag})$;
for $i$ from $0$ to $\log n$ do
    Retrieve $\mathsf{header}_1^i$ and $\mathsf{header}_1^{i+1}$;
    Retrieve $\mathsf{C}_i$ and $\mathsf{C}_{i+1}$ respectively the $i^{th}$ and the $(i+1)^{\mathrm{th}}$
    column of $\mathsf{header}_2^i$ and $\mathsf{header}_2^{i+1}$ of the bucket $\mathcal{P}(\mathsf{tag}, i)$ and
    $\mathcal{P}(\mathsf{tag}, i+1)$;
    // Generate the oblivious permutation $\pi$
    $\pi \xleftarrow{\$} \mathsf{GenPerm}((\mathsf{header}_1^i, \mathsf{C}_i), (\mathsf{header}_1^{i+1}, \mathsf{C}_{i+1}))$;
    // Merge the parent and destination bucket
    $\mathcal{P}(\mathsf{tag}, i+1) = \pi(\mathcal{P}(\mathsf{tag}, i)) + \mathcal{P}(\mathsf{tag}, i+1)$;
end
$B$ =PIR-Read($\mathsf{adr}, \mathcal{P}(\mathsf{tag}, L)$)
```

Algorithm 6: Evict-Clone(adr, tag)

4. C-ORAM ANALYSIS

4.1 C-ORAM correctness analysis

```
Input: Position pos, bucket $\mathcal{P}(\mathsf{tag},\mathsf{level})$, block $B$
Output: Updated bucket $\mathcal{P}(\mathsf{tag},\mathsf{level})$
// Compute the PIR-Write vector $\mathcal{V}$ in client side
for $i$ from $1$ to $\mu \cdot z$ do
    if $i \neq \mathsf{pos}$ then $\mathcal{V}_i \xleftarrow{\$} \mathsf{ENC}(0)$ else
    $\mathcal{V}_i \xleftarrow{\$} \mathsf{ENC}(1)$ ;
end
// Write block in server side
Parse bucket $\mathcal{P}(\mathsf{tag},\mathsf{level})$ as $(\mu \cdot z \times |B|)$ binary matrix $\mathcal{M}$;
$\mathcal{M}_{i,j} = \mathcal{W}_i \cdot B_j$;
$\mathcal{P}(\mathsf{tag},\mathsf{level}) = \mathcal{M} + \mathcal{P}(\mathsf{tag},\mathsf{level})$;
```

Algorithm 7: PIR-Write(pos, block, $\mathcal{P}(\mathsf{tag},\mathsf{level})$), PIR-write process

The goal of the correctness analysis section is to show that, during any eviction namely in Evict and Evict-Clone algorithms, the probability that a failure occurs is very small. The failure in C-ORAM is defined as the lack of encryption of zeros in the evicted path. In this section, we only consider the proof of correctness of C-ORAM's first construction. The proof of correctness of C-ORAM second construction can easily be deduced from the first one. Before starting detailing the main lines of our correctness analysis, we introduce some notations and assumptions.

Let $B_{i,j}$ refer to the bucket at the i^{th} level of the path evicted at the j^{th} step. Each bucket contains $\mu \cdot z$ blocks where μ is a nonnegative integer such that $\mu > 1$. In C-ORAM first construction, the root bucket contains z real elements and $(\mu - 1) \cdot z$ empty blocks. We set $\phi = \mu - 1$. Empty block represents an additively homomorphic encryption of zero. Each bucket cannot have more than z real elements at any time with high probability, see result of Th 4.3. Let $Z_{i,j}$ and $R_{i,j}$ respectively denote the discrete random variables of the number of blocks containing an encryption of zero and the number of real blocks in the bucket $B_{i,j}$ in the i^{th} level and during the j^{th} eviction. Recall that if a real block is pushed to a path leading to a leaf different from its own tag, this block is called a noisy block. Finally, $\tilde{N}_{i,j}$ represents the random variable that counts the number of noisy blocks in the bucket $B_{i,j}$.

Formally, the eviction in Evict and Evict-Clone algorithms fails if $\exists i \in \{0, \cdots, L\}$ and $k \in \mathbb{N}$ such that $Z_{i+1,k} < R_{i,k}$ or $Z_{i,k} < R_{i+1,k}$. Thus, the proof's goal will be to show that there is no integer t that verifies both inequalities with high probability.

First, we introduce two notions that will help us to understand the proof and the eviction mechanism more thoroughly. The first notion is called the *path composition history* while the second one is the *bucket composition history*. Given a path $\mathcal{P}(j)$, the *path composition history* captures the eviction time in which each bucket has been created. Given a bucket $B_{i,j}$, the *bucket composition history* captures all sequence of buckets that have contributed to the construction of the bucket $B_{i,j}$.

Path composition history: In C-ORAM, the eviction follows a deterministic reverse lexicographic order. If we are in the j^{th} step of the eviction, every bucket of the path $\mathcal{P}(j)$ is the result of a previous eviction. Moreover, every bucket in this path has been created from different eviction step.

The buckets belonging to the same path are not arbitrary created. In fact, the time -eviction step- of creation of any bucket follows a pattern. This pattern is an induction relation that links the buckets belonging to the same path. This induction relation links the time of the creation of every bucket depending on the time of creation of other buckets belonging to the same path. For instance, in Fig 4, the path $\mathcal{P}(9)$ of the 9^{th} eviction is composed of the buckets $B_{1,8}$, $B_{2,7}$, $B_{3,5}$ that represent buckets that were respectively created in the 8^{th}, 7^{th} and 5^{th} eviction' step. We do not count the root bucket and the leaf because the pattern of their eviction is clear, namely, the root is evicted every time while the leaf is evicted depending on its reverse lexicographic order.

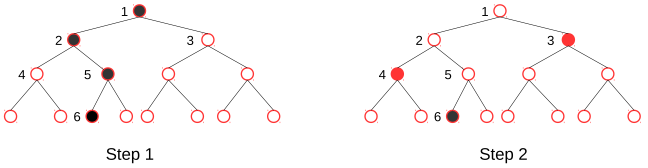


Figure 3: The evicted path contains the buckets in black. The bucket 3 is a copy of the root. Bucket 4 is the result of merging Bucket 1 and 2. Bucket 6 is the result of merging buckets 1, 2 and 5.

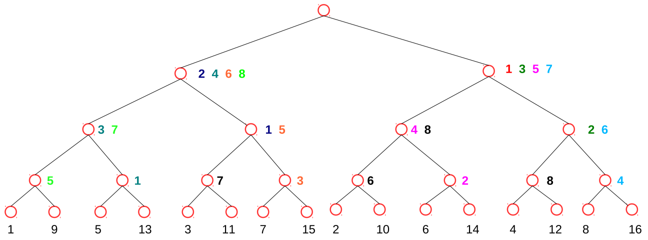


Figure 4: Illustration of nine evictions. The numbers below the leaves represent the order of the reverse deterministic lexicographic eviction. Buckets that have numbers with the same color means that they are evicted at the same step. For example (6,5,3) with the orange color represents the buckets that are evicted in step 7. Numbers 6, 5 and 3 represent the time of creation of these buckets. Black numbers are for buckets that are not yet evicted.

Formally, for N elements stored in the ORAM and $L \in \Theta(\log N)$, the j^{th} eviction, for all $j \geq 1$, is composed of the buckets

$$\{B_{1,j-2^0}, B_{2,j-2^1}, \cdots, B_{L-1,j-2^{L-2}}\}.$$

It is clear from this induction relation that after L operations eviction, all the buckets belonging to an evicted path, except the leaves, are not *new* anymore, i.e., all of them without any exception are copies of a bucket from previous evictions. Thus, in the proof, we will assume that the ORAM has handled a number of eviction larger than L. This will take into account the worst case where all buckets might eventually contain real element as well as noisy ones assuming that beforehand the C-ORAM construction was empty.

Bucket composition history: This notion follows from the previous one. Given a path $\mathcal{P}(j)$, the eviction will empty all buckets in this path except the leaf. The eviction works as follows: the root $B_{0,j}$ will be merged with its destination child $B_{1,j-2^0}$ in the path while the sibling $B'_{1,j}$, originally empty, will be overwritten by a copy of the root, the root is finally overwritten by an empty bucket. The bucket $B_{1,j-2^0}$ will be merged with its destination child $B_{2,j-2^1}$ then emptied. The sibling of the bucket $B_{1,j-2^0}$ will be overwritten by the content of $B_{1,j-2^0}$. We reiterate the process until the end of the path (this was a recapitulation of Evict protocol).

Given a bucket $B_{i,k}$, we want to enumerate all the buckets that resulted in the creation of that bucket. Particularly, we are interested in enumerating the time (step of eviction) of creation of all buckets that contribute on the creation of the bucket $B_{i,k}$. The bucket composition also follows a pattern that is common to any bucket in the construction. Given the eviction algorithm, every

Level$_1$	8	7	6	5	4	3	2	1	-
Level$_2$	7	6	5	4	3	2	1	-	-
Level$_3$	5	4	3	2	1	-	-	-	-
Evicted path	9	8	7	6	5	4	3	2	1

Table 1: Bucket creation pattern in function of the eviction step.

bucket in the i^{th} level is created by merging all the buckets in the path from the root to the $(i-1)^{\text{th}}$ level, see Table 1 for an example of this pattern for $N = 16$. As an instance, the bucket in path 9 at the 3^{rd} level was created during the 5^{th} eviction step. To know what are the buckets that contributed in this bucket creation, we check out the column that has an evicted path equal to 5, then we consider all the buckets that they are in upper levels, namely, buckets 4 and 3 that are respectively in the level 2 and 1.

In general, the bucket $B_{i,j}$ is the result of merging the following buckets:

$$\{B_{0,j}, B_{1,j-2^0}, B_{2,j-2^1}, \cdots, B_{i-1,j-2^{i-2}}\}$$

Now that we have introduced these two observations, we can state our theorem.

THEOREM 4.1. *If the bucket size $z \in \omega(\log N)$, $L \in \Theta(\log N)$ and $\phi \in \Theta(1)$, the probability that $Z_{i+1,j} \geq R_{i,j}$ and $Z_{i,j} \geq R_{i+1,j}$ is in $O(N^{-\log\log N})$, for all $i \in [L]$ and $j \in \mathbb{N}$.*

PROOF. Based on our assumption, we know that a path cannot handle more than z real elements with high probability. This is equivalent to say that $\forall i \in \{0, \cdots, L\}$, we have

$$R_{i+1} + R_i \leq z.$$

To show that $\forall i \in [L]$, $Z_{i+1} \geq R_i$ and $Z_i \geq R_{i+1}$, it is equivalent to show that $\tilde{N}_i \leq \phi \cdot z$. Here, for sake of clarity and without loss of any generality, we do not take into account the eviction step j just to minimize the burden of additional indexes. Then for a given eviction step

$$\begin{aligned} R_{i+1} + R_i &\leq z \\ R_{i+1} + R_i + \tilde{N}_i + Z_i &\leq z + \tilde{N}_i + Z_i \\ R_{i+1} + a \cdot z &\leq z + \tilde{N}_i + Z_i \\ R_{i+1} &\leq (\tilde{N}_i - \phi \cdot z) + Z_i \end{aligned}$$

Therefore, it is sufficient to show that $\tilde{N}_i - \phi \cdot z \leq 0$ in order to proof that $\forall i \in [L]$, $Z_{i+1} \geq R_i$ and $Z_i \geq R_{i+1}$. It is clear that these inequalities hold for any eviction step $j \in [n]$.

In the following, we will proof that the probability that $\tilde{N}_{i,j} > \phi \cdot z$ is negligible with very hight probability.

Based on the bucket composition history pattern, notice that the noisy elements in the bucket $B_{i,j}$ are exactly those that exist already in the bucket $B_{i-1,j-2^{i-2}}$, plus, all the real elements that will be evicted to the other child and therefore they are considered noisy elements for the bucket $B_{i,j}$. Thus, we have $\Pr(\tilde{N}_{i,j} > \phi \cdot z) = \Pr(\tilde{N}_{i-1,j-2^{i-2}} + R'_{i-1,j} > \phi \cdot z)$.

We have shown in the bucket composition history that $B_{i,j}$ is created by summing all the buckets $\{B_{0,k}, B_{1,j-2^0}, B_{2,j-2^1}, \cdots, B_{i-1,j-2^{i-2}}\}$. The above equation can be then formulated more accurately such that $\Pr(\tilde{N}_{i,j} > \phi \cdot z) = \Pr(\max_i(\tilde{N}_{1,j-2^0}, \cdots, \tilde{N}_{i-1,j-2^{i-2}}) + R'_{i-1,j} > \phi \cdot z)$.

The equation can be understood as follows: The noise in bucket $B_{i,j}$ is the maximal amount of noise in any bucket in its history. Each bucket is created independently of the other ones in the evicted path. Therefore the quantity of noise in every bucket in the evicted path is independent of the other ones. Since the noise is cumulative during the eviction, the bucket that has the maximum noise will represent the noise of the last bucket, since based on Algorithm 4 the noise is added up. Also, to this quantity of noise, we add the sum of all real elements in the path that are no longer real elements in the bucket $B_{i,j}$ and therefore represent a new noise represented by $R'_{i-1,j}$.

All buckets in the path are independent of each others, i.e., the number of real elements, the number of noisy elements are independent of the the other buckets in the path. This holds since the real elements, the noise in any bucket is generated from distinct evictions. Therefore we have

$$\begin{aligned} \Pr(\tilde{N}_{i,j} > \phi \cdot z) &= 1 - \Pr(\max_i(\tilde{N}_{1,j-2^0}, \cdots, \tilde{N}_{i-1,j-2^{i-2}}) \\ &\quad + R'_{i-1,j} \leq \phi \cdot z) \\ &= 1 - \prod_{k=1}^{i-1} \Pr(\tilde{N}_{k,j-2^{k-1}} + R'_{i-1,j} \leq \phi \cdot z) \end{aligned}$$

We can reiterate the process of counting the noise until arriving to the root. The quantity of noise in the root is null. Then

$$\begin{aligned} \Pr(\tilde{N}_{i,j} > \phi \cdot z) &= 1 - \prod_{k=1}^{i-1}\prod_{l=1}^{k-1}\cdots\prod_{t=1}^{s-1} \Pr(\tilde{N}_{0,t} + R'_{0,t} + \\ &\quad R'_{1,s} + \cdots + R'_{i-1,j} \leq \phi \cdot z) \\ &= 1 - \prod_{k=1}^{i-1}\prod_{l=1}^{k-1}\cdots\prod_{t=1}^{s-1} \Pr(R'_{0,t} + R'_{1,s} + \\ &\quad \cdots + R'_{i-1,j} \leq \phi \cdot z) \end{aligned}$$

Recall that $R'_{i-1,j}$ represents the number of real elements in the bucket $B_{i-1,j}$ that will be considered as noise in the bucket $B_{i,j}$. Any bucket cannot have more than z elements with hight probability, denoting $\mathcal{R} = R'_{0,t} + R'_{1,s} + \cdots + R'_{i-1,j}$, we then have:

$$\begin{aligned} \Pr(\mathcal{R} \leq \phi \cdot z) &= 1 - \Pr(i \cdot z \geq \mathcal{R} \geq \phi \cdot z) \\ &= 1 - \sum_{k=\phi \cdot z}^{i \cdot z} \Pr(\mathcal{R} = k) \\ &\geq 1 - \sum_{k=\phi \cdot z}^{i \cdot z} \binom{k+i-1}{k} \Pr(R'_{i-1,j} = k) && (2) \\ &\geq 1 - \sum_{k=\phi \cdot z}^{i \cdot z} \binom{k+i-1}{k}\binom{2^{i-1}}{k} \cdot \frac{1}{(2^i)^k} && (3) \\ &\geq 1 - \sum_{k=\phi \cdot z}^{i \cdot z} (\frac{e^2 \cdot (k+i-1)}{2k^2})^k \\ &\geq 1 - (i - \phi) \cdot z \cdot (\frac{e^2 \cdot (\phi \cdot z + i - 1)}{2(\phi \cdot z)^2})^{\phi \cdot z} && (4) \\ &\geq 1 - i \cdot z \cdot (\frac{e^2}{\phi \cdot z})^{\phi \cdot z} && (5) \\ & && (6) \end{aligned}$$

Inequality (1) is the result of calculating all the possible combinations of the equation $x_1 + \cdots + x_k = N$ which is equal to $\binom{k+N-1}{N}$ possibilities. We upper bound this value by multiplying it with the maximum value of probability that equals $\Pr(R'_{i-1,j} = k)$. Inequality (2) is the result of computing the number of real elements that might go to the $(j-1)^{\text{th}}$ level. We have $R'_{j-1,k}$ follows a binomial distribution and then we have $\Pr(R'_{j-1,t} = i) \leq \binom{2^{i-1}}{i} \cdot \frac{1}{(2^i)^k}$. Inequalities (3) and (4) are just an bound that are reached first by replacing $k = \phi \cdot z$ since it will result on the larger value (k is in the denominator) and by summing over the final probability by $i \cdot z$.

Combining all results together we have

$$\begin{aligned}
\Pr(\tilde{N}_{j,k} > \phi \cdot z) &\leq 1 - \prod_{k=1}^{i-1} \prod_{l=1}^{k-1} \cdots \prod_{t=1}^{s-1} (1 - \Pr(j \cdot z \geq \mathcal{R} \geq \phi \cdot z) \\
&\leq 1 - (1 - \Pr(j \cdot z \geq \mathcal{R} \geq \phi \cdot z))^{O(\frac{i^i}{i!})} \\
&\leq 1 - (1 - i \cdot z \cdot (\frac{e^2}{\phi \cdot z})^{\phi \cdot z})^{O(\frac{i^i}{i!})} \\
&= O(\frac{i^i}{i!} iz (\frac{e^2}{\phi \cdot z})^{\phi \cdot z}) \\
&= O(e^i iz (\frac{e^2}{\phi \cdot z})^{\phi \cdot z})
\end{aligned}$$

The last transitions are obtained by the binomial inequality and Stirling approximation. Now, we should define the value of ϕ for which this probability is negligible. The probability above can be simplified to be equal to

$$\Pr(\tilde{N}_{i,j} > \phi \cdot z) = O(e^{i + \ln(i \cdot z) + 2\phi \cdot z - \ln(\phi \cdot z) \cdot \phi \cdot z})$$

This probability computation is independent of the step of eviction $j \in \mathbb{N}$. Therefore, for any $i \in [L]$ we have $i \in \Theta(\log N)$ and $z \in \omega(\log N)$ choosing $\phi \in \Theta(1)$, the probability equals:$\Pr(\tilde{N}_{i,j} > \phi \cdot z) \in O(N^{-\log \log N})$, which is negligible in N.

□

4.2 Security Analysis

4.2.1 Oblivious merging

We prove that permutations generated by Algorithm 4 are indistinguishable from random permutations. Informally, we show that the adversary cannot gain any knowledge about the load of a particular bucket. Applying a permutation from Algorithm 4 is equal to applying any randomly chosen permutation. We formalize our intuition in the security definition below.

First, we introduce our adversarial permutation indistinguishability experiment, that we denote PermG. Let $\mathcal{M}$ denote a probabilistic algorithm that generates permutations based on the configurations of two buckets, and $\mathcal{A}$ a PPT adversary. Let k be the bucket size and s the security parameter. By Perm we denote the set of all possible permutations of size k. Let $\mathcal{E}_1 = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and $\mathcal{E}_2 = (\mathsf{Gen_a}, \mathsf{Enc_a}, \mathsf{Dec_a})$ respectively denote an $\mathsf{IND\$\text{-}CPA}$ encryption and an $\mathsf{IND\text{-}CPA}$ additively homomorphic encryption schemes. $\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s)$ refers to the instantiation of the experiments by algorithm $\mathcal{M}$, $\mathcal{E}_1$, $\mathcal{E}_2$ and adversary $\mathcal{A}$.

The experiment $\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s)$ consists of:

- Generate two keys k_1 and k_2 such that $k_1 \overset{\$}{\leftarrow} \mathsf{Gen_a}(1^s)$ and $k_2 \overset{\$}{\leftarrow} \mathsf{Gen}(1^s)$ and send n buckets additively homomorphic encrypted with $\mathsf{Enc_a}(k_1, .)$ associated to their headers encrypted with $\mathsf{Enc}(k_2, .)$ to the adversary $\mathcal{A}$
- The adversary $\mathcal{A}$ picks two buckets A and B, then sends the encrypted headers $\mathsf{header}(A)$ and $\mathsf{header}(B)$
- A random bit $b \overset{\$}{\leftarrow} \{0,1\}$ is chosen. If $b = 1$, $\pi_1 \overset{\$}{\leftarrow} \mathcal{M}(\mathsf{header}(A), \mathsf{header}(B))$, otherwise $\pi_0 \overset{\$}{\leftarrow} \mathsf{Perm}$. Send π_b to $\mathcal{A}$
- $\mathcal{A}$ has access to the oracle $\mathcal{O}_{\mathcal{M}}$ that issues permutation for any couple of headers different from those in the challenge
- $\mathcal{A}$ outputs a bit b'
- The output of the experiment is 1 if $b' = b$, and 0 otherwise. If $\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, b') = 1$, we say that the adversary $\mathcal{A}$ succeeded.

DEFINITION 4.1 (INDISTINGUISHABLE PERMUTATION). *Algorithm $\mathcal{M}$ generates* indistinguishable permutations *iff for all PPT adversaries $\mathcal{A}$ and all possible bucket configurations of buckets A and B, there exists a negligible function* negl*, such that*

$$\Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s,1) = 1] - \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s,0) = 1] \leq \mathsf{negl}(s).$$

THEOREM 4.2. *If $\mathcal{E}_1$ is $\mathsf{IND\$\text{-}CPA}$ secure, $\mathcal{E}_2$ $\mathsf{IND\text{-}CPA}$ secure, then Algorithm 4 generates indistinguishable permutations.*

PROOF. We consider a succession of games Game_0, Game_1 and Game_2 defined as follows:

- Game_0 is exactly the experiment $\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 1)$
- Game_1 is similar to Game_0, except that encrypted headers are replaced with random strings
- Game_2 is similar to Game_1, except that encrypted buckets are replaced with buckets with new randomly generated blocks which are additively encrypted

From the definition above, we have

$$\Pr[\mathsf{Game}_0] = \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s,1) = 1]. \tag{7}$$

For Game_1, we can construct an efficient distinguisher B_1 that reduces $\mathcal{E}_1$ to $\mathsf{IND\$\text{-}CPA}$ security such that:

$$\Pr[\mathsf{Game}_0] - \Pr[\mathsf{Game}_1] \leq \mathsf{Adv}^{\mathsf{IND\$^-CPA}}_{B_1,\mathcal{E}_1}(s). \tag{8}$$

Similarly for Game_1, we can build an efficient distinguisher B_2 that reduces $\mathcal{E}_2$ to $\mathsf{IND\text{-}CPA}$ security such that:

$$\Pr[\mathsf{Game}_1] - \Pr[\mathsf{Game}_2] \leq \mathsf{Adv}^{\mathsf{IND^-CPA}}_{B_2,\mathcal{E}_2}(s). \tag{9}$$

In the following, we will show that $\Pr[\mathsf{Game}_2] = \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 0) = 1]$. That is, we need to show that the distribution of the output of algorithm $\mathcal{M}$ has a uniform distribution over the set Perm.

For sake of clarity, we assume that the number of noisy slots is zero in both buckets. Therefore, slots in A and B are either full or empty. We can easily extend the proof for the case where we have full, empty and noisy blocks.

For notation clarity, let X denote the discrete random variable that represents the permutation selected by the adversary and by $\mathsf{Load_{i,j}}$ the event of $\mathsf{load}(A) = i$ and $\mathsf{load}(B) = j$. By $\mathsf{load}(A)$, we denote the number of real elements in bucket A. If $b = 0$, the adversary receives a permutation π_0 selected uniformly at random. It is clear that $\mathcal{A}$ cannot distinguish it from another uniformly generated random permutation. Note that in this case, for buckets with k slots, the probability that adversary selects a permutation from Perm uniformly at random equals $\frac{1}{|\mathsf{Perm}|} = \frac{1}{k!}$. Thus, $\Pr[X = \pi_0] = \frac{1}{k!}$.

If $b = 1$, the adversary receives π_1. We need to show that the permutations output by $\mathcal{M}$ are uniformly distributed.

$$\begin{aligned}\Pr(X = \pi_1) &= \sum_{i,j \in [n]} \Pr(X = \pi_1 \text{ and } \mathsf{Load}_{\mathsf{i,j}}) \\ &= \sum_{i,j \in [n]} \Pr(X = \pi_1 \mid \mathsf{Load}_{\mathsf{i,j}}) \cdot \Pr(\mathsf{Load}_{\mathsf{i,j}})\end{aligned}$$

We compute the probability of selecting a permutation while the loads of buckets A and B are fixed to i and j. The number of possible configurations of valid permutations equals $\mathsf{Valid} = \binom{k}{i} \cdot \binom{k-i}{j}$. This represents the number of possible permutation from which the client can choose to generate a valid permutation. From the adversary view, it should take into consideration all possible configurations of blocks in both buckets A and B. The total number of permutations computes to $\mathsf{Total} = \binom{k}{i} \cdot \binom{k}{j} \cdot \binom{k-i}{j} \cdot j! \cdot (k-j)!$. The first two terms count the possible configurations of the loads in both buckets while the three last terms are for valid permutations for a fixed setting of load distribution in the buckets. The cardinality of possible configurations equals the number of possible combinations from which we can select j empty blocks from $k - i$, i.e., $\binom{k-i}{j}$. We then multiply this last value by the possible permutations of the $k - i$ full blocks and the j empty blocks that are respectively equal to $(k - j)!$ and $j!$.

That is,

$$\begin{aligned}\Pr(X = \pi_1 \mid \mathsf{Load}_{\mathsf{i,j}}) &= \frac{\mathsf{Valid}}{\mathsf{Total}} \\ &= \frac{\binom{k}{i} \cdot \binom{k-i}{j}}{\binom{k}{i} \cdot \binom{k}{j} \cdot \binom{k-i}{j} \cdot j! \cdot (k-j)!} \\ &= \frac{1}{\frac{k!}{j!\cdot(k-j)!} \cdot j! \cdot (k-j)!} = \frac{1}{k!}\end{aligned}$$

We plug the result of this equation in the previous one and obtain

$$\Pr(X = \pi_1) = \sum_{i,j \in [n]} \frac{1}{k!} \cdot \Pr(\mathsf{Load}_{\mathsf{i,j}}) = \frac{1}{k!}.$$

Thus for the adversary, permutations output by $\mathcal{M}$ are uniformly distributed, i.e.

$$\Pr[X = \pi_1] = \Pr[X = \pi_0] = \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 0) = 1] \tag{10}$$

Finally, plugging all the results of Equations 7, 8, 9 and 10 together we obtain

$$\begin{aligned}\Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 1)] &= \Pr[\mathsf{Game}_0] \\ &\leq \Pr[\mathsf{Game}_1] + \mathsf{Adv}^{\mathsf{IND\$^-CPA}}_{B_1,\mathcal{E}_1}(s) \\ &\leq \Pr[\mathsf{Game}_2] + \mathsf{Adv}^{\mathsf{IND^-CPA}}_{B_2,\mathcal{E}_2}(s) + \\ &\quad \mathsf{Adv}^{\mathsf{IND\$^-CPA}}_{B_1,\mathcal{E}_1}(s) \\ &\leq \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 0)] + \\ &\quad \mathsf{Adv}^{\mathsf{IND^-CPA}}_{B_2,\mathcal{E}_2}(s) + \mathsf{Adv}^{\mathsf{IND\$^-CPA}}_{B_1,\mathcal{E}_1}(s)\end{aligned}$$

□

4.2.2 Overflow probability of C-ORAM buckets

C-ORAM eviction is similar to Onion ORAM [4]. The distribution of real elements for both constructions is exactly the same. We have a bucket size of $\mu \cdot z$ where z elements are allocated for real elements and $(\mu - 1)z$ is allocated for noisy elements to preserve the correctness of C-ORAM construction. The overflow probability denotes the fact that any bucket in C-ORAM will contain more than z elements. We want to show that this probability is negligible in n. For this, we borrow the results of Devadas et al. [4] and Fletcher et al. [5] that have introduced the *eviction* factor χ. Throughout the paper, we have been stating that $\chi = O(z)$, which is a result of the following theorem, without explicitly stating it before to avoid confusion.

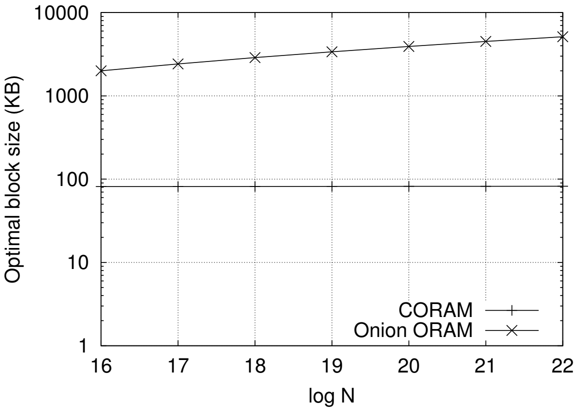


Figure 5: Comparison of block size between C-ORAM and Onion ORAM

THEOREM 4.3. *For the* eviction *factor χ and height L such that $z \geq \chi$ and $N \leq \chi \cdot 2^{L-1}$, the overflow probability after every eviction equals $e^{\frac{-(2z-\chi)^2}{6\chi}}$.*

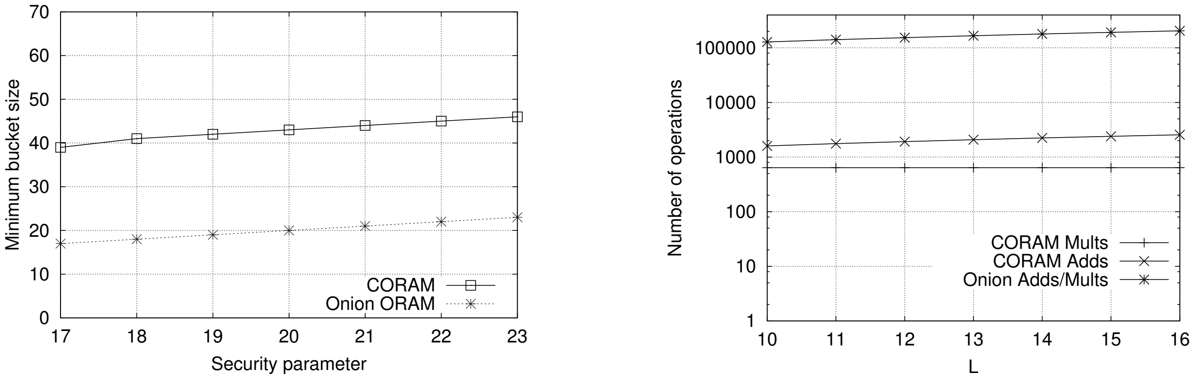
Choosing $z \in \Theta(\lambda)$, $L \in \Theta(\log N)$, $\chi \in \Theta(\lambda)$ and $\lambda \in \omega(\log N)$ makes the the result of Th 4.3 negligible in N.

5. EVALUATION

We have shown analytically that it suffices to set $\mu = \Theta(1)$ and have buckets of size $\Theta(z) = \Theta(\lambda)$. However, we have not derived precisely what bucket size is necessary for concrete security parameters. In order to get an idea of how bucket size in our scheme scales with λ, we performed a series of experiments simulating our ORAM and measuring the maximum number of used slots (real data blocks plus junk blocks) after x number of operations, for various values of x. For instance, if we perform 2^{15} operations on an instance of C-ORAM and the highest bucket load is 35, then we can assure a failure rate of less than 2^{-15} with buckets set to 35. Figure 6a shows the results of this test, compared with Onion ORAMs requirement that buckets be equal to λ.

Additionally we compare the efficiency of our scheme in terms of server computation to that of Onion ORAM. We aim to quantify the number of homomorphic addition and multiplication operations in each scheme, to show that we have significant improvement. Throughout this analysis, we will consider a single multiplication or addition to be over an entire block, although in practice it may be divided into chunks of smaller ciphertext. Any changes in chunk size will apply equally to both schemes so discussion of its impact will be ignored. Note however that we do not have layered encryptions and so, in fact, ciphertext operations in our scheme will be cheaper simply because they are smaller.

During eviction Onion ORAM performs z select operations on each bucket, which each require a PIR query over z slots. This re-



(a) Required bucket size for different security parameters

(b) Comparison of ciphertext operations for Onion ORAM and C-ORAM.

Figure 6: Comparison of C-ORAM and Onion ORAM

sults in a total of z^2 multiplications for each bucket, over L buckets. Each multiplication also implies an addition in the select procedure, so the number of ciphertext additions is the same.

C-ORAM contains one major modifications that is pertinent when comparing ciphertext operations: PIR queries are only done on the root bucket, to add new blocks, and on leaf buckets to read and remove blocks. C-ORAM then requires only $O(z\mu)$ multiplications and $z\mu L$ additions. Since we have shown that μ is a small constant, we effective gain a factor of $O(zL)$ in multiplications and $O(z)$ in additions.

Finally, we compare the optimal block size for C-ORAM in relation to Onion ORAM, cf. Figure 5. For each eviction, Onion ORAM requires $\lambda^2 L$ ciphertexts of size γ to be sent by the client, while we require only permutation vectors of total size $\mu\lambda L \log \lambda$. Since $\gamma = O(\lambda^2)$, this is a huge savings. For reads, Onion ORAM requires $\lambda L\gamma$ bits of ciphertext while we require only $4\mu\lambda\gamma$.

Comparison results: C-ORAM is able to achieve constant communication overhead in the worst-case, with significantly less server computation required in addition to smaller minimum block sizes. Figure 6b shows that we lower both the required number of ciphertext additions and multiplications by several orders of magnitude when compared to Onion ORAM. In exchange for this, we have a slightly higher server storage requirement, increasing by a factor μ. However, Figure 6a shows that this factor is only about 2. Additionally, Figure 5 shows that C-ORAM requires much smaller blocks than Onion ORAM in practice.

6. RELATED WORK

ORAM was first introduced by Goldreich and Ostrovsky [8] and has recently received an increasing interest with the introduction of tree-based ORAM construction [1–3, 5–11, 13, 15, 17, 18, 20, 22–24]. ORAMs can be categorized based on the client memory setting, namely, constant client memory or sublinear client memory. This categorization can be refined by taking into account the server computation nature, namely, storage-only server, versus, computational servers. In the following, we will briefly recapitulate some notable research works done in this area while arranging them in their corresponding categories.

Constant client memory: This category of ORAMs is very useful in the case of very restrained client memory devices such us smartphones, embedded devices. With constant client memory, the aim of this research is to reduce the worst-case or amortized case communication complexity between the client and server [4, 9, 10, 13, 15, 17, 18, 20]. Polylogarithmic amortized-case cost was introduced by Goodrich and Mitzenmacher [9] and Pinkas and Reinman [18] in $O(\log^2 N)$ but with linear worst case communication complexity. This last has been improved to $O(\sqrt{N} \cdot \log^2 N)$ with the work of Goodrich et al. [10]. The first scheme to provide a polylogarithmic worst-case was presented by Shi et al. [20]. The idea behind this scheme is a tree-based construction where nodes consist of small bucket ORAMs, see [8, 16] while memory shuffling is performed after every access. This scheme offers a communication complexity in $O(\log^3 N)$ in term of number of blocks downloaded.

Asymptotics for constant-client memory has been enhanced by the work of Kushilevitz et al. [13] with a communication complexity equal to $O(\frac{\log^2 N}{\log \log N})$. However, this construction suffers from a large hidden constant ~ 30 that make it less efficient compared to Shi et al. [20] for example.

While all the previous schemes are based on storage-only servers, Mayberry et al. [15] have introduced a new paradigm that takes advantage of a computational server setting. In fact, their idea is based on coupling PIR [12] with Shi et al. [20]'s ORAM. A PIR vector is used to retrieve the searched for block from the desired bucket that greatly reduces the amount of bits needed for one access. The authors show therewith that the communication complexity can be reduced to $O(\log^2 N)$.

Devadas et al. [4] enhanced this idea by proposing the first amortized constant client bandwidth in a computational server. The idea is also based on merging PIR and ORAM, however, the client still needs to download a large block size $B = \Omega(\log^5 N)$ which is not very practical for realistic dataset.

Sub-linear client memory Williams and Sion [23], Williams et al. [24] works introduce a sublinear client side memory in $O(\sqrt{N})$ but with a linear worst-case cost complexity. Stefanov et al. [21] improved this result by introducing a polylogarithmic communication complexity in $O(\log^2 N)$ but with $O(\sqrt{N})$ client memory.

Gentry et al. [6] improve the ORAM by Shi et al. [20] by replacing the binary tree by a $\kappa-$array tree. They introduce a new deterministic eviction process adapted to this new structure based on reverse lexicographic ordering of leaves. This eviction method is the basis of many recent tree-based ORAMs such as Fletcher et al. [5] or Devadas et al. [4]. With a branching factor equal to $\kappa = \log N$, the communication complexity of Gentry et al. [6]'s ORAM is in $\frac{\log^3 N}{\log \log N}$. The polylogarithmic client memory is in $O(\log^2 N)$ because the client has to keep track of all elements in path during the eviction.

Stefanov et al. [22] present Path ORAM, one of the most efficient construction with only $O(\log N)$ client memory. The bandwidth is in $O(\log^2 N)$ if the block size is in $\Omega(\log N)$ or in $O(\log N)$ for $\Omega(\log^2 N)$ block sizes. Fletcher et al. [5] further reduced the communication cost by $6 \sim 7\%$.

References

[1] D. Boneh, D. Mazières, and R.A. Popa. Remote oblivious storage: Making oblivious RAM practical, 2011. `http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf`.

[2] K.-M. Chung and R. Pass. A Simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.

[3] I. Damgård, S. Meldgaard, and J.B. Nielsen. Perfectly Secure Oblivious RAM without Random Oracles. In *Proceedings of Theory of Cryptography Conference –TCC*, pages 144–163, Providence, USA, March 2011.

[4] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, and Ling Ren. Onion ORAM: A constant bandwidth and constant client storage ORAM (without FHE or SWHE). *IACR Cryptology ePrint Archive*, 2015:5, 2015.

[5] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A Low-Latency, Low-Area Hardware ORAM Controller with Integrity Verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[6] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Proceedings of Privacy Enhancing Technologies*, pages 1–18, 2013.

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

[8] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43 (3):431–473, May 1996. ISSN 0004-5411. doi: 10.1145/233551.233553. URL `http://doi.acm.org/10.1145/233551.233553`.

[9] M.T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Proceedings of Automata, Languages and Programming –ICALP*, pages 576–587, Zurick, Switzerland, 2011.

[10] M.T. Goodrich, M. Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop –CCSW*, pages 95–100, Chicago, USA, 2011.

[11] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Symposium on Discrete Algorithms –SODA*, pages 157–167, Kyoto, Japan, 2012.

[12] E. Kushilevitz and R. Ostrovsky. Replication is not Needed: Single Database, Computationally-Private Information Retrieval. In *Proceedings of Foundations of Computer Science*, pages 364–373, Miami Beach, USA, 1997.

[13] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Symposium on Discrete Algorithms – SODA*, pages 143–156, Kyoto, Japan, 2012.

[14] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*, pages 314–328, 2005.

[15] T. Mayberry, E.-O. Blass, and A.H. Chan. Path-PIR: Lower Worst-Case Bounds by Combining ORAM and PIR. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, USA, 2014.

[16] R. Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the Symposium on Theory of Computing – STOC*, pages 514–523, Baltimore, USA, 1990.

[17] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *Proceedings of the Symposium on Theory of Computing –STOC*, pages 294–303, El Paso, USA, 1997.

[18] B. Pinkas and T. Reinman. Oblivious ram revisited. In *Advances in Cryptology – CRYPTO*, pages 502–519, Santa Barbara, USA, 2010.

[19] RightScale. State of the cloud report, 2015. URL `http://assets.rightscale.com/uploads/pdfs/RightScale-2015-State-of-the-Cloud-Report.pdf`.

[20] E. Shi, T.-H.H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost. In *Proceedings of Advances in Cryptology – ASIACRYPT*, pages 197–214, Seoul, South Korea, 2011. ISBN 978-3-642-25384-3.

[21] E. Stefanov, E. Shi, and D.X. Song. Towards practical oblivious ram. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, USA, 2012. The Internet Society.

[22] E. Stefanov, M. van Dijk, E. Shi, C.W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of Conference on Computer and Communications Security*, pages 299–310, Berlin, Germany, 2013. ISBN 978-1-4503-2477-9.

[23] P. Williams and R. Sion. Usable pir. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, USA, 2008.

[24] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*, pages 139–148, Alexandra, USA, 2008.