Constant Communication ORAM with Small Blocksize

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

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

Erik-Oliver Blass
Airbus Group Innovations
81663 Munich, Germany
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 two drawbacks. It requires a large block size of $B = \Omega(\log^6 N)$ with large constants. Moreover, while it only needs polylogarithmic computation complexity, that computation consists mostly of expensive homomorphic multiplications. In this work, we address these problems and reduce the required block size to $\Omega(\log^4 N)$. We remove most of the homomorphic multiplications while maintaining $O(1)$ communication complexity. Our idea is to replace their homomorphic eviction routine with a new, much cheaper permute-and-merge eviction which eliminates homomorphic multiplications and maintains 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 server computation.

1. INTRODUCTION

With cloud storage 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 [?] 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 [?]. Building on that, ?] 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, ?] 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, see also [?].

Furthering this research, ?] 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. Their scheme requires that the block has a very large size of $\Omega(\log^6 N)$. In practice, it can be up to 30 MB for reasonably sized databases.
2. 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 tackle 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, as we will show, allows for $O(1)$ communication complexity in the worst case. Finally, we demonstrate via experimental evaluation that our scheme requires only a small storage overhead compared to Onion ORAM. For practical parameter values, we achieve significant improvement in block size and number of homomorphic operations. Table **??** summarizes our improvements when compared to Onion ORAM.

CCS'15, October 12–16, 2015, Denver, CO, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813701 .

Table 1: Comparison of Onion ORAM and C-ORAM, containing block size, worst-case bandwidth, and number of homomorphic additions and multiplications. The simplified block value is a looser bound for easier comparison using $\lambda = \omega(\log N)$ and $\gamma = O(\lambda^3)$.

Scheme	Block size B	Simplified block size	Worst-case bandwidth	# additions	# multiplications
Onion ORAM	$\Omega(\gamma\lambda \log^2 N)$	$\Omega(\log^6 N)$	$O(1)$	$\Theta(B\lambda \log N)$	$\Theta(B\lambda \log N)$
C-ORAM	$\Omega(\lambda[\log \lambda \log N + \gamma])$	$\Omega(\log^4 N)$	$O(1)$	$\Theta(B\lambda \log N)$	$\Theta(B\lambda)$

2. BACKGROUND: ONION ORAM

We start by briefly introducing the main idea of Onion ORAM [**?**] and 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 **?**], 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 [**? ? ?**]. 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 security. Everything is then divided by B, and we get to $O(1)$ communication complexity. Note that $O(1)$ communication complexity 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.

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 ORAM with constant communication complexity and constant client-memory in the number of elements N stored in the ORAM, see Table **??**.

2.2 Analysis

As noted above, $O(1)$ communication complexity does not imply that blocks are the only information exchanged 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.

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 $B \in \Omega(\log N)$ as required by related work [**? ?**]. Generally, large block sizes render ORAMs impractical for many real world scenarios where the block size is fixed and simply predetermined by an application. To mitigate the problem, Onion ORAM uses Lipmaa's PIR [**?**] instead of straightforward additively homomorphic PIR [**?**]. 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 with $\lambda \in \omega(\log N)$. So, it is 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 straightforward approach, the small gain in communication 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.

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 achieves *constant* worst case communication complexity with smaller block size 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 to C-ORAM's main data tree.

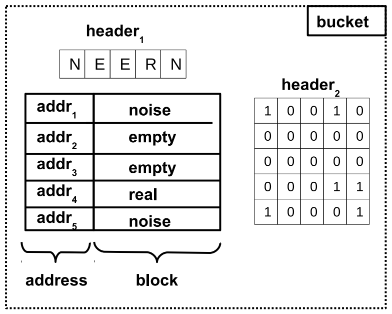


Figure 1: C-ORAM bucket structure

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 $N \leq z \cdot 2^{L-1}$. 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

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 **??**.

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 [**?**] 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 [?].

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 [?]. Note that siblings of this path, after the eviction, will not be empty anymore. See Fig **??** for a sample eviction with $N = 8$.

Sibling buckets, since they 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 $L + 1$ 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 $\mathcal{P}(tag, i)$. Let $[N]$ be the set of integers $\{1, \cdots, N\}$, $x \xleftarrow{\$} [N]$ uniformly sampling a random element from set $[N]$, and χ the period of eviction which is in $O(z)$. $\mathsf{Identity}$ is an empty bucket containing only encryptions of zero.

Algorithm **??** 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 **??** that obliviously retrieves a block. Algorithm **??** shows the eviction where elements percolate towards their leaves using oblivious permutations, see Algorithm **??**.

Block size: The following asymptotic 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., $O(z \cdot L \cdot \log N)$ bits. Also, the client should send a logarithmic number of PIR read vectors $\mathcal{V}$ with size $O(\gamma \cdot z \cdot L)$ bits. Note that the computation of PIR read vectors outputs, for all but one buckets' block, encryption of 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 **??**. Thus, the client only retrieves a single block B. A PIR read applied to all buckets of the path induces an overhead in $O(z \cdot L \cdot \log N + \gamma \cdot z \cdot L + B)$.

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 $O(L \cdot z \cdot \log z)$ 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 $O_z(z \cdot L \cdot \log N + \gamma \cdot z \cdot L + z \cdot \log(z) \cdot L + B)$ communication complexity. To have constant communication complexity in B, the block size should be $B \in \Omega(z \cdot L \cdot \log N + \gamma \cdot z \cdot L + L \cdot z \cdot \log z) \in \Omega(\lambda \cdot \log^2 N + \gamma \cdot \lambda \cdot \log N)$.

The above is a consequence of $z = \Theta(\lambda)$, $\lambda \in \omega(\log n)$, and $L \in \Theta(\log N)$. Based on current attacks [?], $\gamma = O(\lambda^3)$. Therefore, $\lambda \cdot \log^2 N$ is dominated by $\gamma \cdot \lambda \cdot \log N$, and $B \in \Omega(\gamma \cdot \lambda \cdot \log N)$.

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 of 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. Note that we can de-amortize C-ORAM first construction using techniques from [?].

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

Algorithm 1: Access(op, adr, data, ctr, st): C-ORAM access operation, 1st construction

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. Recall that in our first construction, the worst case involves a blowup of $O(z)$, because during eviction the client needs to download $O(z \cdot B)$ bits. In our second and main construction, the eviction remains exactly the same, and our focus will only be on ORAM access.

```
Input: Bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level}))$, address $\mathsf{adr}$
Output: Block $B$
1  Retrieve and decrypt addresses $\mathsf{Addr}$ of bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level}))$;
   // Compute the PIR-Read vector $\mathcal{V}$ in client side
2  if $\mathsf{adr} \in \mathsf{Addr}$ then
       // Retrieve the index $\alpha$
3      $\alpha = \mathsf{Addr}[\mathsf{addr}]$;
4      for $i$ from $1$ to $\mu \cdot z$ do
5          if $i \neq \alpha$ then $\mathcal{V}_i \leftarrow \mathsf{ENC}(0)$ else
6          $\mathcal{V}_i \leftarrow \mathsf{ENC}(1)$ ;
7      end
8  else
9      for $i$ from $1$ to $\mu \cdot z$ do $\mathcal{V}_i \leftarrow \mathsf{ENC}(0)$ ;
10 end
   // Retrieve block in server side
11 Parse bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level})$ as $(\mu \cdot z \times |B|)$ binary matrix $\mathcal{M}$;
12 $B = (\sum_{i=1}^{\mu \cdot z} \mathcal{V}_i \cdot \mathcal{M}_{1,i}, \cdots, \sum_{i=1}^{\mu \cdot z} \mathcal{V}_i \cdot \mathcal{M}_{|B|,i})$;
13 Update $\mathsf{header}_1^{\mathsf{level}}$ of bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level})$;
```
Algorithm 2: $\mathsf{PIR\text{-}Read}(\mathsf{adr}, \mathcal{P}(\mathsf{tag}, \mathsf{level}))$

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. That is, we have to maintain the empty sibling property, as guaranteed by the reverse lexicographic eviction, before evicting any path. Instead of deterministically selecting a path for eviction, we select randomly. However, with 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 resulting 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 our second C-ORAM construction, we are evicting after every access. Consequently, we can be 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 **??** to analyze security and correctness of our modification.

3.3.2 Details and Analysis

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

```
Input: State st
Output: Evicted path and updated state st
1  for $i$ from $0$ to $L-1$ do
2      Retrieve $\mathsf{header}_1^i$ and $\mathsf{header}_1^{i+1}$;
3      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 $\mathcal{P}(\mathsf{st}, i)$ and
       $\mathcal{P}(\mathsf{st}, i+1)$;
4      $\pi \leftarrow \mathsf{GenPerm}((\mathsf{header}_1^i, \mathsf{C}_i), (\mathsf{header}_1^{i+1}, \mathsf{C}_{i+1}))$, generate
       the oblivious permutation $\pi$;
       // Merge the parent and destination bucket
5      $\mathcal{P}(\mathsf{st}, i+1) = \pi(\mathcal{P}(\mathsf{st}, i)) + \mathcal{P}(\mathsf{st}, i+1)$;
6      if $i < L-1$ then
           // Copy the parent bucket into its
              sibling
7          $\mathcal{P}_s(\mathsf{st}, i) = \mathcal{P}(\mathsf{st}, i)$;
8      else
           // Merge the last bucket with the sibling
              leaf
9          Retrieve $\mathsf{header}_1^{i+1}$ and $\mathsf{C}_{i+1}$ from the sibling leaf;
10         $\pi \leftarrow \mathsf{GenPerm}((\mathsf{header}_1^i, \mathsf{C}_i), (\mathsf{header_1}^{i+1}, \mathsf{C}_{i+1}))$;
11         $\mathcal{P}(\mathsf{st}, i+1) = \pi(\mathcal{P}(\mathsf{st}, i)) + \mathcal{P}(\mathsf{st}, i+1)$;
12     end
13     $\mathsf{Update}(\mathsf{header}_1^i)$ and store it with bucket $\mathcal{P}_s(\mathsf{st}, i)$;
14     $\mathsf{Update}(\mathsf{header}_1^{i+1})$ and store it with bucket $\mathcal{P}(\mathsf{st}, i+1)$;
15     $\mathcal{P}(\mathsf{st}, i) = \mathsf{Identity}$;
16 end
```
Algorithm 3: $\mathsf{Evict}(\mathsf{st})$, eviction process

Block size: The access operation in C-ORAM is composed of scheduled path eviction, 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. For sake of clarity, we therefore avoid including them in our asymptotic analysis.

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 writes and single PIR read operation will not change asymptotic 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 $B \in \Omega(z \cdot L \cdot \log z + z \cdot \gamma)$.

With $z \in \Theta(\lambda)$, $\lambda \in \omega(\log N)$ and $L \in \Theta(\log N)$, we achieve $B \in \Omega(\lambda \cdot [\log N \cdot \log \lambda + \gamma])$. In practice, $\gamma \in O(\lambda^3)$, so γ dominates $\log N \cdot \log \lambda$. Therefore, block size B is $B \in \Omega(\gamma\lambda)$.

Our second C-ORAM construction achieves worst-case constant blow-up and omits inefficient PIR reads performed for ORAM access. This second construction improves the block size by a multiplicative factor of $\log^2 N$ compared to Onion ORAM.

As you can see, the main overhead of C-ORAM's block size comes from the size of ciphertext γ. Recall that $\gamma \in O(\lambda^3)$. Therefore, the smaller the additively homomorphic ciphertext will get, the smaller the block size of C-ORAM will be.

4. C-ORAM ANALYSIS

4.1 C-ORAM correctness analysis

The goal of the correctness analysis section is to show that, during any eviction (algorithms Evict and Evict-Clone), the probability that a failure occurs is small. The failure in C-ORAM is defined

```
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 $\pi$
1  Let $x_1, x_2$ be the number of empty and noisy slots in $A$;
2  Let $y_1, y_2$ be the number of full and noisy slots in $B$;
3  $d_1 = x_1 - y_1$;
4  $d_2 = x_2 - y_2$;
5  for $i$ from 1 to $\mu \cdot z$ do
6      case $B[i]$ is full  $z \xleftarrow{\$}$ all empty slots in $A$ ;
7      case $B[i]$ is noisy
8          if $d_2 > 0$ then
9              $z \xleftarrow{\$}$ all noisy slots in $A$;
10             $d_2 = d_2 - 1$;
11         else
12             $z \xleftarrow{\$}$ all empty slots in $A$;
13         end
14     end
15     case $B[i]$ is empty
16         if $d_1 > 0$ then
17             $z \xleftarrow{\$}$ all non-assigned slots in $A$;
18             $d_1 = d_1 - 1$;
19         else
20             $z \xleftarrow{\$}$ all full slots in $A$;
21         end
22     end
23     $\pi[i] = z$;
24     $A[z] =$ assigned;
25 end
26 return $\pi$;
```
Algorithm 4: GenPerm(A, B), oblivious permutation generation

```
Input: Operation op, address adr, data data, state st
Output: Block $B$ associated to address adr
// Fetch tag value from position map
1  tag = posMap(adr);
2  posMap(adr) $\xleftarrow{\$} [N]$;
// Retrieve desired block
3  $B$ =Evict-Clone(adr, tag);
4  if op = write then  set $B$ = data ;
// Select a random position in the root bucket
5  $\mathsf{pos}_1 \xleftarrow{\$} [\mu \cdot z]$;
// Write back the block to the empty root
6  PIR-Write($\mathsf{pos}_1, B, \mathcal{P}(\mathsf{st}, 0)$);
7  Evict(st);
// Select a random noisy block position from
   the header of the leaf $\mathcal{P}(\mathsf{st}, L)$
8  $\mathsf{pos}_2 \xleftarrow{\$} \mathsf{header}^L$;
9  $N$ =PIR-Read($\mathsf{pos}_2, \mathcal{P}(\mathsf{st}, L)$);
// Write back the negation of the noisy block
10 PIR-Write($\mathsf{pos}_2, -N, \mathcal{P}(\mathsf{st}, L)$);
```
Algorithm 5: Access(op, adr, data, st): C-ORAM access operation, 2nd construction

```
Input: Leaf tag and address adr
Output: Block $B$
1  Create a copy of the C-ORAM path $\mathcal{P}(\mathsf{tag})$;
2  for $i$ from 0 to $L-1$ do
3      Retrieve $\mathsf{header}_1^i$ and $\mathsf{header}_1^{i+1}$;
4      Retrieve $\mathsf{C}_i$ and $\mathsf{C}_{i+1}$ respectively the $i^{th}$ and the $(i+1)^{\text{th}}$
       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 $\pi$
5      $\pi \leftarrow$ GenPerm(($\mathsf{header}_1^i, \mathsf{C}_i$), ($\mathsf{header}_1^{i+1}, \mathsf{C}_{i+1}$));
       // Merge the parent and destination bucket
6      $\mathcal{P}(\mathsf{tag}, i+1) = \pi(\mathcal{P}(\mathsf{tag}, i)) + \mathcal{P}(\mathsf{tag}, i+1)$;
7  end
8  $B$ =PIR-Read(adr, $\mathcal{P}(\mathsf{tag}, L)$);
9  for $i$ from 0 to $L$ do
10     Update $\mathsf{header}_1^i$ in $\mathcal{P}(\mathsf{tag}, i)$;
11 end
```
Algorithm 6: Evict-Clone(adr, tag)

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's second construction is a straightforward extension from the first one. Before presenting details 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, with integer $\mu > 1$. In C-ORAM's first construction, the root bucket contains z real elements and $(\mu-1)\cdot z$ empty blocks. We set $\phi = \mu - 1$. An empty block represents an additively homomorphic encryption of zero. Each bucket cannot have more than z real elements at any time with high probability, as we will prove in Theorem **??**. Let $Z_{i,j}$ be the discrete random variable of the number of blocks containing an encryption of zero in bucket $B_{i,j}$. Similarly, $R_{i,j}$ represents the number of real blocks. 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. $\tilde{N}_{i,j}$ represents the random variable that counts the number of noisy blocks in bucket $B_{i,j}$. Finally, the j^{th} eviction step is the eviction of the j^{th} path following a deterministic reverse lexicographic order.

Formally, the eviction in Evict algorithm fails if $\exists i \in \{0, \ldots, L\}$ 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 such integer $i \in \{0, \ldots, L\}$ that verifies both inequalities with high probability.

First, we need to introduce two properties that will help us to understand the proof and the eviction mechanism more thoroughly. The first property 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 step in which each bucket has been created. Given a bucket $B_{i,j}$, the *bucket composition history* is a sequence that captures all 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. In the j^{th} step of eviction, every bucket of the path $\mathcal{P}(j)$ has been created on a previous eviction. Thus, we associate to a bucket its *eviction step* during which it has been created. In particular, every bucket in this path has been created from a different eviction step. We are interested on defining the relation between the eviction steps of buckets belonging to the same evicted path. This relation follows a pattern which is common to all evicted paths. For instance, in Fig. **??**, the path $\mathcal{P}(9)$ of the 9th eviction is composed of buckets $B_{1,8}, B_{2,7}, B_{3,5}$. These are buckets that were created, respectively, in the 8th, 7th, and 5th eviction step. We do not count the root bucket and the leaf, because the pattern of their eviction is clear. That is, the root is evicted every time while the leaf is evicted following reverse lexicographic order.

Formally, for N elements stored in the ORAM and $L \in \Theta(\log N)$, one can easily show by induction that the j^{th} evicted path, for all $j \geq 1$, is composed of $\{B_{1,j-2^0}, B_{2,j-2^1}, \ldots, B_{L-1,j-2^{L-2}}\}$.

After L evictions, buckets belonging to an evicted path, except the leaves, are copies of a bucket from previous evictions. In our proof, we will later assume that the ORAM has performed a num-

```
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
1 for $i$ from $1$ to $\mu \cdot z$ do
2 |   if $i \neq$ pos then $\mathcal{V}_i \leftarrow \mathsf{ENC}(0)$ else
3 |   $\mathcal{V}_i \leftarrow \mathsf{ENC}(1)$ ;
4 end
  // Write block in server side
5 Parse bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level})$ as $(\mu \cdot z \times |B|)$ binary matrix $\mathcal{M}$;
6 $\mathcal{M}_{i,j} = \mathcal{W}_i \cdot B_j$;
7 $\mathcal{P}(\mathsf{tag}, \mathsf{level}) = \mathcal{M} + \mathcal{P}(\mathsf{tag}, \mathsf{level})$;
```

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

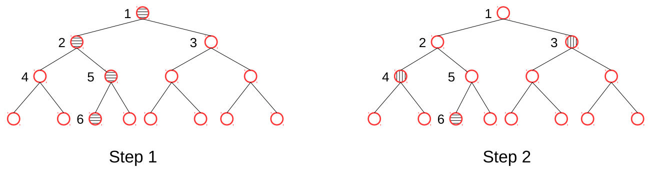


Figure 2: Buckets on evicted path are with horizontal hatching. Bucket 3 is a copy of the root. Bucket 4 results from merging buckets 1 and 2. Bucket 6 results from merging 1, 2, and 5.

ber of evictions larger than L. We will also consider the worst case where all buckets might eventually contain real or noisy blocks.

Bucket composition history: This property 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).

Given a bucket $B_{i,j}$, we are interested in enumerating the eviction's steps of creation of all buckets that have contributed to bucket $B_{i,j}$. The bucket composition also follows a pattern that is unique to any bucket in the construction. Given the eviction algorithm, every bucket in the i^{th} level is created by merging all buckets in the path from the root to the $(i-1)^{\text{th}}$ level, see Table **??** for an example of this pattern for $N = 16$. As an example, the bucket in path 9 at the 3rd level was created during the 5th eviction step. To determine

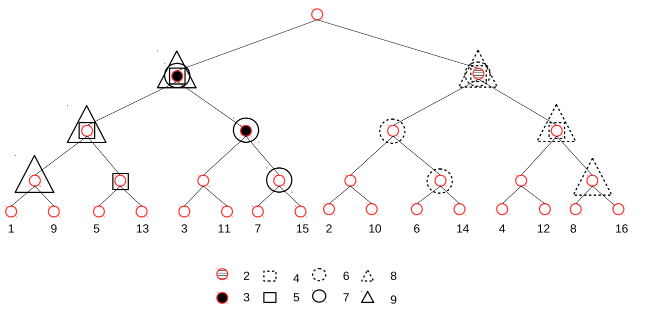


Figure 3: Illustration of nine evictions. Numbers below leaves represent the order of reverse deterministic lexicographic eviction. Buckets with same shapes were full and then evicted at the same step. Example: buckets with triangular shape are evicted in step 9.

Table 2: Bucket creation pattern in function of the eviction step.

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

the buckets that contributed to this bucket's creation, we check the column that has an evicted path equal to 5. Then, we consider all buckets that are in upper levels: buckets 4 and 3 which are in levels 2 and 1. In general, a bucket $B_{i,j}$ is the result of merging the following buckets: $\{B_{0,j}, B_{1,j-2^0}, B_{2,j-2^1}, \ldots, B_{i-1,j-2^{i-2}}\}$.

Noisy blocks: It is important to understand the source of provenance of noisy blocks. From a one hand, a noisy block can be created whenever an access has been performed on C-ORAM. Therefore, the accessed block is not valid anymore and should be turned to a noisy block by updating the headers. On the other hand, a noisy block can be also created from the eviction process. During an eviction, and in particular, when a parent is copied to its sibling, many real elements are no longer valid and become noisy. The main goal of this section is to upper bound the number of noisy blocks in all buckets with high probability. Our quantification has then to take into account both sources, however, one can show that the first source of noisy blocks can be included as a worst case of the second source. Recall that a bucket cannot hold more than z real elements which means that we can have up to z real elements turning to noise –if we access the same bucket z times before eviction–. One can only add z additional blocks to each bucket to handle reads, so whatever computed bound on ϕ, one can increase it by one. However, one can show that it is not necessary. In fact, this situation is equivalent to having all z real elements in a given bucket as noise for its sibling (which is the worst case). Thus, one can consider the first source of noisy blocks as a sub-case of the second one. We are now ready to state our main theorem.

THEOREM 4.1. *If $\phi \in \Theta(1)$, the probability that $Z_{i+1,j} \geq R_{i,j}$ and $Z_{i,j} \geq R_{i+1,j}$ is in $O(z^{-z})$, 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 implies that $\forall i \in \{0, \ldots, L\}$, we have $R_{i+1} + R_i \leq z$.

Here, for sake of clarity and without loss of any generality, we omit the eviction step j from notation just to minimize the burden of additional indexes. 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 $\tilde{N}_i \leq \phi \cdot z$:

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

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 $j \in [N]$.

Consequently, the proof boils down to show that the probability that $\tilde{N}_{i,j} > \phi \cdot z$ is negligible with very high probability.

Based on the bucket composition history, 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 $\{B_{0,j}, B_{1,j-2^0}, B_{2,j-2^1}, \ldots, B_{i-1,j-2^{i-2}}\}$. The above equation can be then formulated more accurately such that $\Pr(\tilde{N}_{i,j} > \phi \cdot z) = \Pr(\max_i(\tilde{N}_{1,j-2^0}, \ldots, \tilde{N}_{i-1,j-2^{i-2}}) + R'_{i-1,j} > \phi \cdot z)$.

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. We give more details below about the independence assumption. Since the noise is cumulative during the eviction, the bucket that has the maximum noise will represent the noise of the last bucket. Recall that based on Algorithm **??**, the noisy blocks are added up. Also, to this quantity of noise, we add the sum of all real elements in the path that are no longer considered real elements in $B_{i,j}$ and therefore represent a new noise denoted by $R'_{i-1,j}$.

All buckets in an evicted 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. First, note that a bucket is created by merging all its ancestors. We have defined in the previous section the notion of *bucket composition history* that keeps track of each bucket's ancestor that contributed to its creation. As have been shown, the bucket ancestors emanate from different evictions' steps which is a consequence of the reverse deterministic lexicographic eviction. More importantly, each bucket in the evicted path has a different bucket composition history such that the intersection of all of them is *empty*. Formally, an evicted path, based on the *path composition history*, of the j^{th} step equals $\{B_{1,j-2^0}, B_{2,j-2^1}, \ldots, B_{L-1,j-2^{L-2}}\}$.

Consider a bucket and its parent in the evicted path for $i \in \{1, \ldots, L-1\}$, $B_{i,j-2^{i-1}}$ and $B_{i+1,j-2^i}$. The bucket composition is $\{B_{0,j-2^{i-1}}, B_{1,j-2^{i-1}-2^0}, B_{2,j-2^{i-1}-2^1}, \ldots, B_{i-1,j-2^{i-1}-2^{i-2}}\}$ and $\{B_{0,j-2^i}, B_{1,j-2^i-2^0}, B_{2,j-2^i-2^1}, \ldots, B_{i,j-2^i-2^{i-1}}\}$.

By replacing each bucket in the above sequence by its own bucket composition history and by iterating the process, we will converge to a state where each bucket is composed of root buckets that were instantiated at different evictions' steps. That is, no distinct buckets in the evicted path have a root in common. Thus, the number of real and noisy elements are independent from each other. Therefore,

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

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

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

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 $\mathcal{R} = R'_{0,t} + R'_{1,s} + \ldots + R'_{i-1,j}$, we then have

$$\begin{aligned}\Pr(\mathcal{R} \leq \phi \cdot z) &= 1 - \Pr(i \cdot z \geq \mathcal{R} \geq \phi \cdot z) \\ &= 1 - \sum_{k=\phi \cdot z}^{i \cdot z} \Pr(\mathcal{R} = k) \qquad (1)\end{aligned}$$

Now, we have to compute an upper bound of $\Pr(\mathcal{R} = k)$. One can proceed by: (1) counting all possible solutions of $\mathcal{R} = k$, then (2) multiply this value by the probability of the most likely solution. All the possible combinations of the equation $x_1 + \ldots + x_k = N$ equal $\binom{k+N-1}{N}$ possibilities. The non-trivial part is to find an upper bound of the most likely solution that, in its general form, equals $\Pr(R'_{0,t} = x_1 \text{ AND } \ldots \text{ AND } R'_{i-1,j} = x_{i-1})$. We have that R'_{i-1,x_i} follows a binomial distribution such that $\Pr(R'_{i-1,t} = x_i) \leq \binom{2^{i-1}}{x_i} \cdot \frac{1}{(2^i)^{x_i}}$. Using the independence between buckets, see above for the independence argument, we obtain:

$$\begin{aligned}\Pr(\bigwedge_{j=0}^{i-1} \Pr(R'_j = x_j)) &= \prod_{j=0}^{i-1} \Pr(R'_j = x_j) \leq \prod_{j=0}^{i-1} \binom{2^{j-1}}{x_j} \cdot \frac{1}{(2^j)^{x_j}} \\ &\leq \prod_{j=0}^{i-1} (\frac{e}{2x_j})^{x_j} = (\frac{e}{2})^k \prod_{j=0}^{i-1} (\frac{1}{x_j})^{x_j}\end{aligned}$$

We want to find a readable upper bound only in function of k. Also, remark that the above inequality is true iff, $\forall j \in \{1, \ldots, i-1\}$, the following statement holds $x_j \leq 2^{j-1}$. One can verify with induction that with x_j's reaching their upper bounds 2^{j-1} minimizes $\prod_{j=1}^{i-1} x_j^{x_j}$ and therefore maximizes the inverse of the function. Also, there exists by construction a nonnegative integer γ such that $\sum_{i=1}^{\gamma-1} 2^i \leq k \leq \sum_{i=1}^{\gamma} 2^i$, which implies that $\gamma - 1 \leq \log k \leq \gamma + 1$. Putting everything together we obtain:

$$\begin{aligned}\Pr(\bigwedge_{j=0}^{i-1} \Pr(R'_j = x_j) &= (\frac{e}{2})^k \prod_{j=0}^{i-1} (\frac{1}{x_j})^{x_j} \\ &\leq (\frac{e}{2})^k \cdot \frac{1}{2^2 \cdot (2^2)^{2^2} \cdots (2^{\gamma-1})^{2^{\gamma-1}}} \\ &= (\frac{e}{2})^k \cdot \frac{1}{2^{\sum_{j=1}^{\gamma-1} j 2^j}} = (\frac{e}{2})^k \cdot \frac{1}{2^{1+(\gamma-2)2^\gamma}} \\ &\leq (\frac{e}{2})^k \cdot \frac{1}{2^{1+(\log k - 3)k}} \leq (\frac{4e}{k})^k\end{aligned}$$

Now, we plug the above results in (**??**)

$$\begin{aligned}\Pr(\mathcal{R} \leq \phi \cdot z) &\geq 1 - \sum_{k=\phi \cdot z}^{i \cdot z} \binom{k+i-1}{k} (\frac{4e}{k})^k \\ &\geq 1 - \sum_{k=\phi \cdot z}^{i \cdot z} (\frac{4e^2 \cdot (k+i-1)}{k^2})^k \\ &\geq 1 - (i - \phi) \cdot z \cdot (\frac{8e^2 \cdot (\phi \cdot z + i - 1)}{2(\phi \cdot z)^2})^{\phi \cdot z} \qquad (2) \\ &\geq 1 - i \cdot z \cdot (\frac{8e^2}{\phi \cdot z})^{\phi \cdot z} \qquad (3)\end{aligned}$$

Inequalities (**??**) and (**??**) are bounds 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

$$\begin{aligned}\Pr(\tilde{N}_{j,k} > \phi \cdot z) &\leq 1 - \prod_{k=1}^{i-1}\prod_{l=1}^{k-1}\cdots\prod_{t=1}^{s-1}(1 - \Pr(j \cdot z \geq \mathcal{R} \geq \phi \cdot z)) \\ &\leq 1 - (1 - \Pr(j \cdot z \geq \mathcal{R} \geq \phi \cdot z))^{O(\frac{i^i}{i!})} \\ &\leq 1 - (1 - i \cdot z \cdot (\frac{8e^2}{\phi \cdot z})^{\phi \cdot z})^{O(\frac{i^i}{i!})} \\ &= O(\frac{i^i}{i!} iz(\frac{e^2}{\phi \cdot z})^{\phi \cdot z}) = O(e^i iz(\frac{e^2}{\phi \cdot z})^{\phi \cdot z})\end{aligned}$$

The last transitions are obtained by the binomial inequality and Stirling approximation. Now, we define the value of ϕ for which this probability is negligible. The probability above can be simplified 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, choosing $\phi \in \Theta(1)$ (and assuming that $L \in O(z \ln z)$), the probability equals: $\Pr(\tilde{N}_{i,j} > \phi \cdot z) \in O(e^{-z \ln z})$, which is negligible in z. $\square$

COROLLARY 4.1. *If 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 $j \in \mathbb{N}$.*

The Corollary can be derived from the main theorem by taking $z \in \omega(\log N)$.

4.2 Security Analysis

4.2.1 Oblivious merging

We prove that permutations generated by Algorithm **??** 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 **??** 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 $k_1 \xleftarrow{\$} \mathsf{Gen_a}(1^s)$ and $k_2 \xleftarrow{\$} \mathsf{Gen}(1^s)$ 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 $b \xleftarrow{\$} \{0, 1\}$ is chosen. If $b = 1$, $\pi_1 \xleftarrow{\$} \mathcal{M}(\mathsf{header}(A), \mathsf{header}(B))$, otherwise $\pi_0 \xleftarrow{\$} \mathsf{Perm}$. 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 $\mathcal{A}$ succeeded.

DEFINITION 4.1 (INDISTINGUISHABLE PERMUTATION). *Algorithm $\mathcal{M}$ generates* indistinguishable permutations *iff for all PPT adversaries $\mathcal{A}$ and all possible 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 **??** generates indistinguishable permutations.*

PROOF. We consider a sequence of games (Game_0, Game_1, 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

$$\Pr[\mathsf{Game}_0] = \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 1) = 1]. \quad (4)$$

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

$$|\Pr[\mathsf{Game}_0] - \Pr[\mathsf{Game}_1]| \leq \mathsf{Adv}^{\mathsf{IND\$^-CPA}}_{B_1,\mathcal{E}_1}(s). \quad (5)$$

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

$$|\Pr[\mathsf{Game}_1] - \Pr[\mathsf{Game}_2]| \leq \mathsf{Adv}^{\mathsf{IND^-CPA}}_{B_2,\mathcal{E}_2}(s). \quad (6)$$

We will no 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 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.

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

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,

$$\Pr(X = \pi_1 \mid \mathsf{Load_{i,j}}) = \frac{\mathsf{Valid}}{\mathsf{Total}}$$
$$= \frac{\binom{k}{i} \cdot \binom{k-i}{j}}{\binom{k}{i} \cdot \binom{k}{j} \cdot \binom{k-i}{j} \cdot j! \cdot (k-j)!} = \frac{1}{\frac{k!}{j! \cdot (k-j)!} \cdot j! \cdot (k-j)!} = \frac{1}{k!}$$

We insert the result of this equation in the previous one and obtain $\Pr(X = \pi_1) = \sum_{i,j \in [n]} \frac{1}{k!} \cdot \Pr(\mathsf{Load_{i,j}}) = \frac{1}{k!}$.

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

$$\Pr[X = \pi_1] = \Pr[X = \pi_0] = \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 0) = 1] \quad (7)$$

Combining Equations **??**, **??**, **??**, and **??**, we obtain

$$\begin{aligned}
&\Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 1)] = \Pr[\mathsf{Game_0}] \\
&\leq \Pr[\mathsf{Game_1}] + \mathsf{Adv}^{\mathsf{IND\$^-CPA}}_{B_1,\mathcal{E}_1}(s) \\
&\leq \Pr[\mathsf{Game_2}] + \mathsf{Adv}^{\mathsf{IND^-CPA}}_{B_2,\mathcal{E}_2}(s) + \mathsf{Adv}^{\mathsf{IND\$^-CPA}}_{B_1,\mathcal{E}_1}(s) \\
&\leq \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 0)] + \mathsf{Adv}^{\mathsf{IND^-CPA}}_{B_2,\mathcal{E}_2}(s) + \mathsf{Adv}^{\mathsf{IND\$^-CPA}}_{B_1,\mathcal{E}_1}(s).
\end{aligned}$$

□

4.2.2 Overflow probability of C-ORAM buckets

C-ORAM eviction is similar to Onion ORAM [**?**]. 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 **?**] and **?**] 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.

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 **??** 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. We performed 20 sets of runs for each value of the security parameter where we executed 2^λ operations to test security parameter λ. For each of these runs, we measured the largest bucket load in the tree and then averaged this value across all runs to determine a bucket size which matches the selected security parameter. Figure **??** shows the results of this test, compared with equivalent tests run using the original Onion ORAM algorithm. Our results show that, because of the lower value of A in C-ORAM, our bucket size is actually slightly smaller than Onion ORAM.

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.

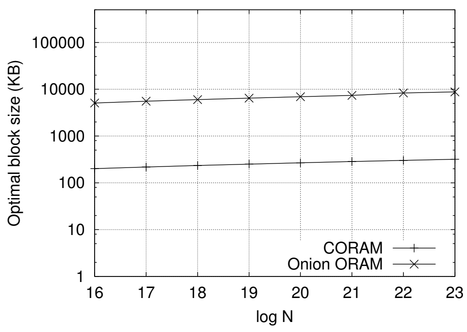
During eviction Onion ORAM performs z select operations on each bucket, which each require a PIR query over z slots. This results in a total of z^2 multiplications for each bucket, over L buckets. Amortized over z gives $O(z \cdot L)$ multiplications. Each multiplication also implies an addition in the select procedure, so the number of ciphertext additions is the same.

C-ORAM contains one major modification 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 $O(z \cdot \mu)$ multiplications and $z \cdot \mu \cdot L$ additions. Since we have shown that μ is a small constant, we effective gain a factor of $O(L)$ in multiplications. Crucially, this means the number of multiplications for C-ORAM is independent of $\log N$, as can be seen in Figure **??**.

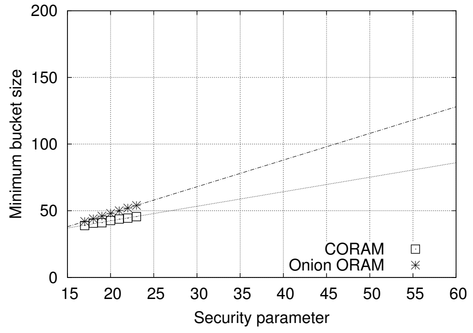
Figure **??** shows the results of tests we have run to determine the computational speedup of C-ORAM compared to Onion ORAM. We considered Pailler encryption as the homomorphic cipher for our tests, using a 2048-bit semiprime, which results in ciphertexts of size 4096 bits. Tests were done on a 2013 Macbook Pro with a 2.5 GHz Intel Core i7 processor, which we found could perform 62 ciphertext multiplications per second. We then calculated how much time it would take to perform the necessary ciphertext operations for one ORAM access, setting $B = 100kb$ and varying N from 2^{16} to 2^{23}. Figure **??** also shows the computation necessary for the online (read) portion of the access.

Although C-ORAM improves significantly over Onion ORAM, computation is the main bottleneck in both schemes. C-ORAM requires less than one MB of communication for one of the queries we tested. Using modern Internet connections, communication would take only a matter of seconds compared to a minute for the ciphertext computations. C-ORAM takes about 7 minutes for this query, while Onion ORAM takes over an hour and a half.

We stress that these evaluation results are largely to show the relative improvement of C-ORAM over Onion ORAM. We chose Paillier, because it is an established additively homomorphic encryption scheme, with well-understood levels of security. There are new homomorphic encryption schemes which perform much better than Paillier [**? ? ?**]. But stable, optimized implementations of them do not yet exist, and concrete parameters choices are still up for debate. Preliminary tests indicate that use of, for instance, the modern NTRU encryption scheme of **?**] could allow

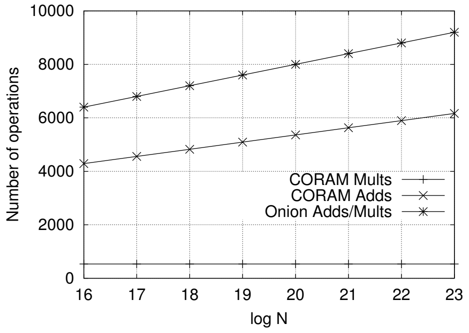


(a) Minimum efficient block size for C-ORAM and Onion ORAM

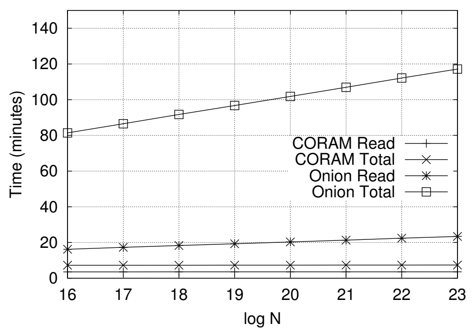


(b) Required bucket size in relation to security parameter

Figure 4: Comparison of C-ORAM and Onion ORAM



(a) Required ciphertext operations for one access



(b) Comparison of computation time for one access

Figure 5: Comparison of C-ORAM and Onion ORAM

for accesses with as little as 5 to 10 seconds of computation. However, a significant drawback that must be balanced for NTRU is that the ciphertexts are much larger, resulting in a tradeoff between increased computational efficiency and higher communication for the PIR portions of C-ORAM. We leave full exploration of optimized homomorphic encryption schemes to future work.

Finally, we compare the optimal block size for C-ORAM in relation to Onion ORAM, cf. Figure **??**. 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 $\gamma = O(\lambda^3)$, 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 **??** shows that we lower both the required number of ciphertext additions and multiplications by several orders of magnitude when compared to Onion ORAM, and Figure **??** shows that in practice this leads to a substantial improvement in efficiency. Figure **??** shows that, due to our lower value of A, the bucket size for C-ORAM is actually smaller in practice than Onion ORAM as well. Additionally, Figure **??** shows that C-ORAM requires much smaller blocks than Onion ORAM in practice.

6. RELATED WORK

ORAM was first introduced by **?**] and has recently received an increasing interest with the introduction of tree-based ORAM construction [**? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?**]. 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 [**? ? ? ? ? ? ? ?**]. Polylogarithmic amortized-case cost was introduced by **?**] and **?**] 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 **?**]. The first scheme to provide a polylogarithmic worst-case was presented by **?**]. The idea behind this scheme is a tree-based construction where nodes consist of small bucket ORAMs, see [**? ?**] 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 **?**] with a communication complexity equal to $O(\frac{\log^2 N}{\log\log N})$. However, this construction suffers from a large hidden constant $\sim$ 30 that make it less efficient compared to **?**] for example.

While all the previous schemes are based on storage-only servers, **?**] have introduced a new paradigm that takes advantage of a computational server setting. In fact, their idea is based on coupling PIR [**?**] with **?**]'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 $O(\log^2 N)$. **?**] 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 $B = \Omega(\log^6 N)$ which is not very practical for realistic dataset.

Sub-linear client memory: **?** **?**] works introduce a sublinear client side memory in $O(\sqrt{N})$ but with a linear worst-case cost complexity. **?**] improved this result by introducing a polylogarithmic communication complexity in $O(\log^2 N)$ but with $O(\sqrt{N})$ client memory.

?] improve the ORAM by **?**] 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 **?**] or **?**]. With a branching factor equal to $\kappa = \log N$, the communication complexity of **?**]'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. **?**] 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. **?**] further reduced the communication cost by 2 to 4 times.

Acknowledgements: We thank shepherds Chris Fletcher, Ling Ren, and Elaine Shi for valuable discussions improving the paper. This work was partially supported by NSF grant 1218197.