CH^{f}-ORAM: A Constant Communication ORAM without Homomorphic Encryption

Tarik Moataz
Colorado State University
Telecom Bretagne
tarik.moataz@colostate.edu

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

Travis Mayberry
United States Naval Academy
mayberry@usna.edu

ABSTRACT

Recent techniques reduce ORAM communication complexity down to constant in the number of blocks N. However, they induce expensive homomorphic encryption on both the server and the client. In this paper, we present an alternative approach CH^{f}-ORAM. This ORAM features constant communication complexity without homomorphic encryption, in exchange for expanding the traditional ORAM setting from single-server to multiple non-colluding servers. We show that adding as few as 4 servers allows for substantially reduced client and server computation compared to existing single-server alternatives. Our approach uses techniques from information-theoretically secure Private Information Retrieval to replace homomorphic encryption with simple XOR operations. Besides $O(1)$ communication complexity, our construction also features $O(1)$ client memory and a block size of only $\Omega(\log^{3} N)$. This leads to an ORAM which is extremely lightweight and suitable for deployment even on memory and compute constrained devices. Finally, CH^{f}-ORAM features a circuit size which is constant in the blocksize making it especially attractive for secure RAM computations.

1. INTRODUCTION

The classic measure of ORAM efficiency is its communication overhead. Research has recently achieved optimal $O(1)$ communication complexity in the number of blocks stored in the ORAM [11, 28]. Unfortunately, these schemes achieve theoretically optimal communication complexity by leveraging substantial server computation that can become prohibitively expensive in practice. In current schemes [28], additively homomorphic encryption like Paillier [31] is used on the server to perform oblivious shuffling. The resulting server computation exceeds several minutes for every client access which (in terms of time) often outweighs savings in communication complexity. Other recent research results [9, 38] demonstrate how to leverage multiple servers to decrease communication complexity to $O(1)$ between client and servers. Yet, this comes at the additional cost of either (1) additional communication overhead between servers, (2) a weaker security model trusting servers or (3) additional client memory overhead rendering it impractical on resource-constrained devices.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

Our contribution: Towards a practical, lightweight, and low latency ORAM, our goal is an ORAM having constant communication complexity without homomorphic encryption. We follow a setup similar to Stefanov and Shi [38] where we relax the single server assumption for ORAMs and allow a small number (as few as 4) of non-colluding servers. This allows us to replace expensive encryption by a very efficient XOR approach. Based on this, we build a new ORAM with optimal $O(1)$ communication complexity *and* efficient client and server computations on the order of milliseconds instead of minutes. Additionally, our ORAM uses the traditional client-server model and does not need extra communication between servers. As a result, we enable ORAM applications with lightweight clients that might not even be able to encrypt, but only to communicate and perform a few XOR operations.

Generally, it is straightforward to adapt recent tree-based ORAMs and remove encryption by leveraging multiple servers. Encryption can be replaced by, e.g., secret sharing. However, this comes with no reduction in client-server communication complexity or client storage complexity — a contribution of this paper. Also, in contrast to Stefanov and Shi [38], we show how to take advantage of a multi-server setting to decrease communication complexity of recent efficient tree-based ORAMs.

To summarize, we present a new multi-server ORAM (CH^{f}-ORAM) with the following *technical highlights*:

- constant communication complexity without expensive additively homomorphic encryption. Computation is ca. $30\times$ faster than C-ORAM [28] and ca. $300\times$ faster than Onion ORAM [11] in a setting with $N = 2^{20}$ blocks.
- a block size in $\Omega(\log^{3} N)$, which is smaller by a factor of $\log N$ than C-ORAM's [28] and $\log^{2} N$ smaller than Onion ORAM's [11].
- an eviction circuit size constant in block size B, for $B \in \widetilde{\Omega}(\log^{3} N)$.
- in contrast to Stefanov and Shi [38], there is no inter-server communication required.

By not requiring inter-server communication, we avoid situations where extra cost due to this inter-server communication might outweigh savings from constant client-server communication.

We stress that security of our construction relies only on a non-collusion assumption, but no computational assumption.

Table 1 compares CH^{f}-ORAM to related work.

1.1 Motivation

Expensive Homomorphic Encryption: To access one block obliviously in C-ORAM [28], only a constant (in N) number of blocks are retrieved from the server. This is possible due to computational

Table 1: Comparison of recent ORAMs, block size B in function of the number of blocks N, client memory in blocks, and communication complexity in blocks.

Scheme	Block Size B	Client Memory	Communication	# homomorphic scalar multiplications	# Servers	Inter server communication
Ring ORAM [35]	$\Omega(\log^2 N)$	$O(\log N)$	$O(\log N)$	–	1	–
Onion ORAM-AHE [11]	$\Omega(\log^5 N)$	$O(1)$	$O(1)$	$\Theta(B\lambda\log N)$	1	–
C-ORAM [28]	$\Omega(\log^4 N)$	$O(1)$	$O(1)$	$\Theta(B\lambda)$	1	–
SS [38]	$\Omega(\log^2 N)$	$O(\sqrt{N})$	$O(1)$	–	2	$O(\log N)$
LO [28]	$\Omega(\log N)$	$O(1)$	$O(\log N)$	–	2	–
CH^f-ORAM	$\Omega(\log^3 N)$	$O(1)$	$O(1)$	–	4	–

power leveraged on the server. However, the number of homomorphic and additive multiplications on the server side adds non-trivial delay to the communication. Thus, even if the communication attains its lower bound, the computation might annihilate advantages in communication.

As an example, we take results from C-ORAM's evaluation [28]. Using Pailler as their additive homomorphic encryption, for $N = 2^{20}$, the computation in C-ORAM takes around 10 minutes for one ORAM operation. This does not even take the time into account that is required to generate CPIR queries – these are again based on a homomorphic encryption of a logarithmic number of vectors. We conjecture that the overall delay per operation exceeds the above 10 minutes by far, motivating the need for a lightweight ORAM.

Secure RAM Computations: Using ORAM for secure RAM computation was introduced by Ostrovsky and Shoup [30]. Circuit-based secure multi-party computation inherently depends on the size of the inputs, making it inefficient for large inputs. Also, transforming RAM programs to circuits turns out to be expensive. If $O(T)$ is the running time of a program, transforming it first to a Turing machine leads to a $O(T^3)$ running time Turing machine [8]. Then, transforming the Turing machine to a circuit will lead to a $O(T^3 \cdot \log T)$ circuit size [33]. Using an ORAM for secure RAM computation, Gordon et al. [21] show how a compilation can be performed in a two-party computation based on Shi et al. [37]'s ORAM. A binary search program has then been made affordable in a poly-log time complexity rather than linear time as required by circuit based solutions. Similar to the setting of Lu and Ostrovsky [25] that have shown how to leverage a two-servers ORAM for secure party computation, we propose CH^f-ORAM as a much more suitable construction for secure RAM computation. In fact, with a constant communication complexity in blocks B, secure RAM computation becomes much cheaper, with the only additional requirement of having four parties instead of two and having blocks larger than a specific threshold—It generalizes to $2k$ parties for any $k \geq 2$. We give more details about secure RAM computation over CH^f-ORAM in Section 6.

Non-colluding Servers: By non-colluding servers we mean that all servers are fully malicious, but the are not allowed to share their state or coordinate in an attack. With four servers, we envision four different adversaries, each controlling one server.

At first it may seem like a significant disadvantage that our scheme requires multiple servers and they must be trusted not to collude. However, consider that one of the most promising scenarios for ORAM is adding access pattern privacy to cloud services. In this scenario it is quite easy and reasonable to find non-colluding servers. There are many different cloud providers competing with each other in this sector (Amazon, Microsoft, Google, Apple, Dropbox, etc.) and they are economically motivated not to share information with each other. Additionally, the threat of outside hackers or insider threats compromising multiple cloud providers simultaneously, each with their own independent infrastructure, is likely very low.

2. BACKGROUND

We briefly present a high-level overview of tree-based ORAMs, focusing on the constant communication ORAM by Moataz et al. [28]. We also give an overview of the IT-PIR by Chor et al. [5].

2.1 Tree-based ORAM

An ORAM allows two operations on outsourced memory. Read(a) reads from and Write(a,data) writes to a block of data given the memory address a and the new data to be written. A tree-based ORAM stores N blocks of data in a binary tree with N leaves. Each node in the tree is a "trivial" ORAM bucket[1], typically accessed as a whole. Each bucket contains $\lambda \in O(\log N)$ blocks, giving a failure (overflow) probability of $2^{-\lambda}$ during later eviction. Each leaf is associated to a leaf tag tag $\in \{0,1\}^{\log N}$. To access an element, the user keeps a position map that maps an ORAM address to its leaf tag. The size of the position map is in $O(N \cdot \log N)$. To allow client memory to be constant in N, the position map is stored on the server as an ORAM, too. This results in a recursive ORAM structure, where access to the position map requires accessing $O(\log N)$ ORAMs of increasing size. After resolving leaf tag tag for an address, the desired block resides on the path $\mathcal{P}(\mathsf{tag})$ between the root of the tree and leaf tag. A Read and Write in an tree-based ORAM with constant client memory are often realized as a ReadAndRemove operation which additionally removes the block from the tree. This is then followed by an Add operation. Finally, to prevent the root bucket from overflowing, an eviction process is necessary to percolate real blocks towards their tagged leaves.

C-ORAM [28]: C-ORAM is a tree-based ORAM with the following specifics. First, blocks are encrypted with an additive homomorphic encryption. Every bucket is prepended by IND-CPA encrypted "headers" containing information about the bucket's contents such as block addresses, block tags as well as which slots in the bucket are empty. The ReadAndRemove, Add, and Evict operations are implemented as follows:

- ReadAndRemove(a): Given address a, the client fetches leaf tag tag from the position map. Given tag, the client downloads all headers in the path $\mathcal{P}(\mathsf{tag})$ that starts from the root and ends with leaf tag. The client decrypts the headers in $\mathcal{P}(\mathsf{tag})$, then sends a computational private information retrieval (C-PIR) query to privately retrieve the block a. The header is updated to indicate that the slot a was found in is now empty. Finally, the client re-encrypts all buckets' headers and uploads them to the path $\mathcal{P}(\mathsf{tag})$. This downloading, re-encrypting, and uploading a path is performed on a block-by-block basis to keep client memory constant.

[1]We use bucket and node interchangeably in this paper.

- $\mathsf{Add(a,data)}$: The client randomly samples a new leaf tag t from $\{0,1\}^{\log N}$, updates the position map, and encrypts the block with the new data. The client then adds the block to the root with a C-PIR write operation.
- $\mathsf{Evict}(\nu)$: The client downloads all encrypted headers from path $\mathcal{P}(\nu)$, where ν is chosen based on a reverse lexicographic ordering of the leaves. It then generates a set of L permutations using an *oblivious merge* algorithm (see below) that will merge every parent-child pair on the path. The client also copies the contents of each bucket in the path to its sibling and updates the headers for all buckets appropriately.

Oblivious Merge: Given two C-ORAM buckets, each containing a number of "empty", "real", and "noisy" blocks, an oblivious merge algorithm [28, 29] outputs a permutation that merges both buckets into one. Noisy blocks are real blocks that are either percolated to the "wrong" path as a result of an eviction or previously leftover from a read operation. The permutation arranges blocks of the first bucket such that no real blocks in the first bucket are at the same position as real blocks in the second bucket. The server can then use the additively homomorphic feature of the encryption and add the blocks at the same position in the two buckets. This results in a single combined bucket, where no real blocks are lost. The important property of this merge is that, for the adversary, the permutation is computationally indistinguishable from a randomly chosen permutation on buckets. Informally, the adversary does not learn any information about bucket contents. The communication cost of a permutation is low for scenarios such as path eviction in [28].

The permutation is generated such that real blocks in one of the buckets are only ever lined up with empty blocks in the other bucket. Empty blocks are initialized as encryptions of 0, so a real block with data x homomorphically added to an empty block results in a ciphertext that encrypts $x + 0 = x$. Therefore, if the server permutes the blocks in the parent bucket according to the client's instruction, the resulting combined bucket will have all the real blocks from both buckets. Noisy blocks can be safely lined up with other noisy blocks since their values are not important. Additively homorphic encryption is crucial for this step because it allows the server to merge buckets without the client processing the contents of those buckets. This is the property that allows the reduction from $O(\log N)$ communication complexity to $O(1)$.

Complexity Analysis: ORAMs have constant communication complexity if the total amount of data exchanged between client and server is asymptotically constant in the number of blocks exchanged. Informally, per read from or write to a block in an ORAM, there is only a constant number of blocks exchanged between client and server. That is, all headers and permutations transmitted during operations or evictions are in $O(B)$. Moataz et al. [28] show that the total size of headers and permutation is in $O(\log^4 N)$, so this is achieved when choosing $B \in \Omega(\log^4 N)$.

2.2 IT-Secure Private Information Retrieval

Information-theoretically secure Private information retrieval (IT-PIR) is a cryptographic primitive introduced by Chor et al. [5]. In contrast to ORAM, IT-PIR does not require the outsourced database to be encrypted to hide access patterns. However, only read operation are possible in IT-PIR. To additionally achieve information-theoretic security, some form of database redundancy is required. For example, the database needs to be replicated to k servers, and these servers must not collude with each other. There exists a large body of work improving communication complexity of the initial construction, e.g., see [2, 3, 22] and many derivatives. For completeness sake, we also mention that there exists work preserving information theoretic security against up to $t < k$ colluding servers [12].

In this paper, we are particularly interested in retrieving blocks of data, not a single bit. This makes our choice of PIR scheme easier because the size of the blocks are actually relatively large in relation to the number of choices for each PIR selection ($O(\log N)$ blocks per bucket). It turns out that the basic construction by Chor et al. [5] is sufficient for the needs of our CH^f-ORAM construction.

Chor et al. [5] overview: The client wants to retrieve block data stored at position pos out of a sequence of N blocks. Therefore, the client starts by generating a random bit vector vect_1 of length N bit and sends it to Server 1. A second vector vect_2 is the same as vect_1, only the bit at position pos is flipped; vect_2 is sent to Server 2. Each server XORs all blocks where the corresponding bit in the vector is set to 1. The final result is sent back to the client. The client can restore data by XORing each server's output. Note that the communication complexity of this information-theoretically secure PIR is linear in N. The construction is shown in Algorithm 1.

```
   Input: Position $\mathsf{pos} \in \{1,...,N\}$
   Output: Block $\mathsf{data}$ stored at $\mathsf{pos}$
   // Client, generate PIR vectors
1  Set $\mathsf{adr}[\mathsf{pos}] = 1$ and $\forall i \neq \mathsf{pos} : \mathsf{adr}[i] = 0$;
2  $\mathsf{vect}_1 \xleftarrow{\$} \{0,1\}^N$;
3  $\mathsf{vect}_2 := \mathsf{Vect}_1 \oplus \mathsf{adr}$;
4  Send $\mathsf{vect}_1$ to Server 1 and $\mathsf{vect}_2$ to Server 2;
5  $\mathsf{rsl}_1 = \mathsf{rsl}_2 = 0^N$;
6  for $i$ from 0 to $N$ do
       // Server 1
7      if $\mathsf{vect}_1[i] = 1$ then set $\mathsf{rsl}_1 = \mathsf{rsl}_1 \oplus B_i$ ;
       // Server 2
8      if $\mathsf{vect}_2[i] = 1$ then set $\mathsf{rsl}_2 = \mathsf{rsl}_2 \oplus B_i$ ;
9  end
10 Send $\mathsf{rsl}_1$ and $\mathsf{rsl}_2$ to client;
   // Client
11 $\mathsf{data} := \mathsf{rsl}_1 \oplus \mathsf{rsl}_2$;
```

Algorithm 1: Linear IT-PIR for two servers

Private Block Retrieval: Several improved IT-PIR constructions exist with better communication overhead. For example, Beimel et al. [3] offer $O(N^{\frac{\log\log k}{k \cdot \log k}})$. However, most of IT-PIR constructions (with linear property) can be transformed into an information-theoretic private block retrieval that can recover a block of B bits in $O(B)$. Formally, let IT-PIR$(1,N,k)$ be an IT-PIR that reads one bit using k servers with N bit stored in each. Let IT-PIR(B,N,k) be an IT-PIR that reads a block of B bit using k servers with N blocks stored in each. That is, each server stores $N \cdot B$ bit in total.We capture this claim in the following corollary:

COROLLARY 2.1. *(Corollary 12 of [5]) There exists an IT-PIR*(B, N,k) *construction with* B *times the complexity of IT-PIR*$(1, \frac{N}{B}+1, k)$.

Consequently, if the block size B is larger enough, we can achieve optimal constant communication complexity. This is captured by the following result:

THEOREM 1. *(Corollaries 13 and 14 of [5]) For any constant* $k \geq 2$ *and for any* $B \geq k \cdot N$, *there exists an IT-PIR*(B, N, k) *construction with communication complexity* $O(B)$ *bit.*

Clearly, if the number of blocks N is replaced by a poly-logarithmic number of blocks, then the block size can be therefore also poly-logarithmic in N to provide a constant IT-PIR block retrieval.

2.3 Secret Sharing

For CH^f-ORAM, we also need secret sharing. Informally, secret sharing enables a dealer to share a secret among a number of parties such that no one alone can recover the secret. The parties can recover the secret only by joining their shares. There are many schemes with different properties and approaches, e.g., [4, 36], but again we will only focus on a basic form of secret sharing. Given a secret S, the dealer generates a random string S_1 and a string S_2 such that $S = S_1 \oplus S_2$. S_1 and S_2 represent the shares to be distributed to two parties.

3. CH^f-ORAM

With CH^f-ORAM, we combine C-ORAM with IT-PIR. CH^f-ORAM inherits C-ORAM's major properties such as its eviction technique, bucket size, tree structure, and main aspects of the oblivious merge technique.

Technical Challenge: Replacing homomorphic encryption is challenging. While IT-PIR reads can easily be used and replace (homomorphic encryption) PIR reads, changes to the ORAM, i.e., writes and eviction require more attention. First, the oblivious merge must be adopted to our "no-encryption" idea. In C-ORAM, the oblivious merge makes use of additively homomorphic encryption which we have to find a suitable secure equivalent without encryption for.

Second, in C-ORAM, adding a new block to the root is simple, i.e., the client uses a PIR read followed by a PIR write to overwrite a stale block in the root. In CH^f-ORAM, we propose a new *batch insertion* which replaces the PIR write operation.

Overview: Consider a version of the C-ORAM tree where buckets and headers are unencrypted. Let this unencrypted ORAM tree be Tree. We create two shares from Tree, Tree_1 and Tree_2, such that $\mathsf{Tree} = \mathsf{Tree}_1 \oplus \mathsf{Tree}_2$, bucket by bucket, block by block, and header by header. We store the two shares at two *non-colluding* servers s_1 and s_2.

Trivially, the above splitting gives security: without collusion, a server cannot learn anything from its share. As we will be using IT-PIR on each share separately, we need to introduce two more non-colluding servers, s'_1 and s'_2, that replicate the shares. In conclusion, share Tree_1 is stored at servers s_1, s'_1, respectively, and Tree_2 is stored at servers s_2,s'_2, respectively.is a lower

Towards even more practicality, we also reduce the block size. Our idea is to leverage a new hybrid model: we fetch the position map separately using Path ORAM [41] adapted to the multi-server setting. This reduces the block size by a factor of $\log N$ while keeping constant communication complexity in total.

3.1 Details

CH^f-ORAM is a tree-based ORAM of height L. Each bucket contains 3 headers header^1, header^2, and header^3, and λ blocks of size B bit. The first two headers are λ by L two dimensional arrays. Each row i in header^1 contains the address of block i in that bucket. Each row i in header^2 contains the leaf tag of block i in that bucket. header^3 is a λ by 2 bit matrix that captures the state of every block in the bucket. Each block can be either empty, real or noisy. For details about the meaning of empty, real or noisy, refer to Section 2.

Note that the height L is slightly smaller than $\log N$, the bucket size λ is in $O(\log N)$, and the noisy blocks are a consequence of using the oblivious merge technique [28]. We will give more details about the choice of these parameters in the analysis, cf. Section 4. For each bucket and header, we compute 2 shares. A block data in a bucket is equal to $\mathsf{data} = \mathsf{data}_1 \oplus \mathsf{data}_2$, with $\mathsf{data}_1 \overset{\$}{\leftarrow} \{0, 1\}^B$. We store data_1 on two servers s_1,s'_1, respectively, and data_2 on servers s_2, s'_2, respectively. For ease of exposition, whenever we mention *downloading* or *uploading* a header/bucket, this implicitly refers to retrieving the corresponding shares using IT-PIR construction, cf. Algorithm 1. For ease of exposition, assume that the position map is stored on the client side. We show later that recursively outsourcing the position map in smaller Path ORAM trees [41] (instead of CH^f-ORAM trees) would save a logarithmic factor to the block size.

To access (read or write) a block at address adr, the client first fetches leaf tag tag from the position map. Second, the client instructs servers to download all bucket headers on the path $\mathcal{P}(\mathsf{tag})$. For each header, the client receives back 4 different bit strings b_1, b_2,b_3,b_4 and reassembles the header by computing $b_1 \oplus b_2 \oplus b_3 \oplus b_4$. Using the header, the client knows the exact position of the block at address adr on path $\mathcal{P}(\mathsf{tag})$. That is, the client knows which bucket and which block in that bucket.

Now, the client generates a random $\lambda \cdot L$ bit IT-PIR query vector $\mathsf{vect}_1 \overset{\$}{\leftarrow} \{0,1\}^{\lambda \cdot L}$. The client also creates a second $\lambda \cdot L$ bit vector vec_2 that is equal to vect_1 besides that the bit at position adr is flipped: $\mathsf{vect}_2[\mathsf{pos}] = 1 \oplus \mathsf{vect}_1[\mathsf{adr}]$. The client sends vector vect_1 to servers s_1,s'_1 and vect_2 to servers s_2,s'_2.

Each server performs the conditional XOR operation described in the IT-PIR computation of Algorithm 1 and sends the resulting bit string back to the client. To recover the block at address adr, the client computes an XOR over all 4 strings received. Finally, the client modifies the corresponding header of the block and marks it as noisy.

The client adds the (changed in case of a write) block back to the root. For this, the client first downloads all headers, generates new shares while updating both the header of the root and the header of the block that has been accessed. Also, as with C-ORAM, one block from the leaf of path $\mathcal{P}(\mathsf{tag})$ has to be refreshed in case it contains a noisy block. This is performed in a deterministic manner. For the same leaf, a block is never accessed twice before accessing all other blocks.

The challenge is now not to let the root bucket overflow, i.e., we need to evict. Remember that in C-ORAM, the client could use PIR writes to obliviously place a block in any position within the root. Together with oblivious permutations, the whole eviction then becomes oblivious, too. Without a PIR write (requiring expensive operations), however, the servers would be able to observe all client modifications to the root and tracing new real blocks added.

A first idea for eviction would be that the client waits χ operations and then evicts real blocks from the root bucket as follows. The client downloads the entire root bucket, shuffles the position of real elements and adds a number of empty blocks. The client then uploads the shuffled root bucket to the servers. Finally, the client downloads headers, computes the oblivious merge algorithm, and outputs L oblivious merge permutations for the servers. The servers will use these permutations to merge every two adjacent (parent-child) buckets from the root down to the leaf. The order of the eviction follows a reverse deterministic lexicographic order.

As this eviction is performed after every χ read operations, we achieve *amortized* constant communication complexity: given a bucket size λ and $\chi \in O(\lambda)$, too. The challenge is now to de-amortize the eviction such that also worst-case communication complexity is in $O(1)$.

That implicitly requires inserting the real block after every access while introducing some randomness, as the root bucket will not be entirely shuffled.

Batch insertion: To de-amortize the above eviction, we introduce the concept of batch insertions that works as follows. First, note that after every evict, the root bucket is empty. Now, instead of

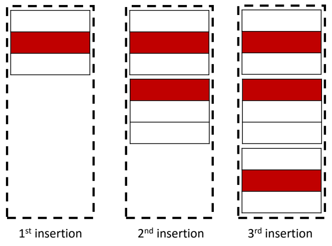

Figure 1: Batch insertion in the root bucket for $\phi = 3$ and $\chi = 3$.

inserting the real block after an access operation and shuffling the entire root bucket on the client side, we upload a total of ϕ blocks after every access. That is, we upload the real block together with $\phi - 1$ empty blocks that will be used to handle noisy blocks in lower levels of the tree.

Throughout the batch insertion, the root load will increase by ϕ blocks after every access. After χ accesses, the bucket size equals $\chi \cdot \phi$. At this point, we obtain a root bucket which contains χ real blocks and $\chi \cdot (\phi - 1)$ empty blocks. As long as ϕ is constant, the cost of inserting a block in the root is constant in B, see Figure 1 for an illustration of the batch insertion.

Once the root bucket's size equals $\chi \cdot \phi$, the client needs to instruct the server to evict the root bucket following a reverse lexicographic order. The eviction cost, in this case, is evenly distributed over χ accesses. The communication complexity of the eviction is now reduced by $\frac{1}{\chi}$ multiplicative factor, and is now equal to a single access operation. We give more details about the eviction communication complexity in Section 3.3. From a security perspective, the way how the real and empty blocks are inserted in the root bucket gives some additional information to the server, such as no two real blocks exist on the same ϕ inserted blocks. In prior C-ORAM [28], the PIR write does not give this public information to the server. These additional information disclosed by our batch insertion technique have a considerable impact on the security of the permutations generated by the client, and will lead to a different and quite more sophisticated proof that we will detail in Section 4.

We present pseudo-code about access operation and eviction in Algorithms 2 and 3.

3.2 Correctness

We now discuss "correctness" of CH^f-ORAM. That is, we explain why the read operation (Algorithm 2) will really output the block in question data. Second, we show how merging buckets in Algorithm 3 preserves the values of real blocks. As writes are only adding data to the root bucket, they are trivially correct.

3.2.1 Read Operation

In Algorithm 2, lines 17-20, the server performs XOR operations over blocks for which the IT-PIR vector contains a 1. We know that CH^f-ORAM's tree is stored over four servers. Two servers are storing two shares, while the other two store exact duplicates of each share. To retrieve a block, the client retrieves shares from all four servers.

Specifically, the first server stores for each block data_i a random value r_i such that the second server stores $\mathsf{data}_i \oplus r_i$. The third and forth server store, respectively, duplicates of r_i and $\mathsf{data}_i \oplus r_i$. Now for each read operation, path $\mathcal{P}(\mathsf{pos})$ has to be accessed in order to retrieve the desired block, see Algorithm 2 lines 1-2. To identify the bucket that contains the wanted block and the position of the block in this bucket, the client first downloads all headers (lines 3-11). Given the position, the client generates four IT-PIR

```
   Input: Operation op, address adr, data block, counter ctr, state st
   Output: Block $B$ associated to address addr
   // Fetch tag value from position map
1  tag = posMap(adr);
2  posMap(adr) $\overset{\$}{\leftarrow} [N]$;
3  for $i$ from 0 to $L$ do
4      download $\mathsf{header}_i^1$ and $\mathsf{header}_i^3$ ;
5      for $j$ from 1 to $\lambda$ do
           // Search if adr exists in $\mathsf{header}^1$
6          if $\mathsf{header}_i^1[j] = \mathsf{adr}$ then
7              set pos $:= i \cdot \lambda + j$;
               // Update headers
8              set $\mathsf{headers}_i^1[j] = 0$ and $\mathsf{headers}_i^3[j] = \mathsf{noisy}$;
9          end
10     end
11 end
   // Generate the PIR vectors
12 for $i$ from 0 to $L \cdot \lambda$ do
       // $\delta_{i,j}$ is the Dirac function
          equal to 1 if $i = j$ and 0 otherwise
13     $\mathsf{ptr}[i] := \delta_{i,\mathsf{pos}}$;
14 end
15 $\mathsf{vect}_1 \overset{\$}{\leftarrow} \{0,1\}^{L \cdot \lambda}$;
16 $\mathsf{vect}_2 := \mathsf{Vect}_1 \oplus \mathsf{ptr}$;
   // Computation on servers $s_1$ and $s'_1$
17 for $i$ from 0 to $L \cdot \lambda$ do
18     if $\mathsf{vect}_1[i] = 1$ then set $\mathsf{rsl}_{1,1} := \mathsf{rsl}_{1,1} \oplus \mathcal{P}(\mathsf{tag}, i)$ ;
19     if $\mathsf{vect}_2[i] = 1$ then set $\mathsf{rsl}_{2,1} := \mathsf{rsl}_{2,1} \oplus \mathcal{P}(\mathsf{tag}, i)$ ;
20 end
   // Computation on client side.
      $\mathsf{rsl}_{1,2}$ and $\mathsf{rsl}_{2,2}$ are the output of similar
      computation, as above, by servers $s_2$ and $s'_2$
21 $\mathsf{data} := \mathsf{rsl}_{1,1} \oplus \mathsf{rsl}_{2,1} \oplus \mathsf{rsl}_{1,2} \oplus \mathsf{rsl}_{2,2}$;
22 if op = write then set data = block ;
   // Batch insertion
23 upload new shares
   of data, and $\phi - 1$ empty blocks to the root bucket in a random order;
   // Refresh headers
24 upload new shares for all headers;
25 if ctr = 0 mod $(\chi)$ then Evict(st) ;
26 set ctr := ctr + 1;
```

Algorithm 2: Access(op, adr, block, ctr, st): CH^f-ORAM access operation

```
   Input: Eviction state st
1  for $i$ from 0 to $L - 1$ do
2      download headers $\mathsf{H_i} = \{\mathsf{header}_i^1, \mathsf{header}_i^2, \mathsf{header}_i^3\}$
       and $\mathsf{H}_{i+1}$ of bucket $\mathcal{P}(\mathsf{st}, i)$ and $\mathcal{P}(\mathsf{st}, i+1)$;
       // generate oblivious merge permutation
3      set $\pi \leftarrow \mathsf{GenPerm}(\mathsf{H_i}, \mathsf{H}_{i+1})$;
       // Merge the parent and destination bucket
4      $\mathcal{P}(\mathsf{st}, i+1) := \pi(\mathcal{P}(\mathsf{st}, i)) \oplus \mathcal{P}(\mathsf{st}, i+1)$;
5      if $i < L - 1$ then
           // Copy
              the parent bucket into its sibling
6          $\mathcal{P}_s(\mathsf{st}, i) := \mathcal{P}(\mathsf{st}, i)$;
7      else
           // Merge
              the last bucket with the sibling leaf
8          download headers $\mathsf{H}_{i+1}^{\mathsf{s}}$ from the sibling leaf;
9          $\pi \leftarrow \mathsf{GenPerm}(\mathsf{H_i}, \mathsf{H}_{i+1}^{\mathsf{s}})$;
10         $\mathcal{P}(\mathsf{st}, i+1) := \pi(\mathcal{P}(\mathsf{st}, i)) \oplus \mathcal{P}(\mathsf{st}, i+1)$;
11     end
12     update and upload new shares of headers $\mathsf{H_i}$ and $\mathsf{H}_{i+1}$;
13     set $\mathcal{P}(\mathsf{st}, i) := \mathbf{0}^{\lambda \cdot B}$;
14 end
```

Algorithm 3: Evict(st): CH^f-ORAM evict operation

vectors (lines 15-16) and sends to the corresponding servers. For two servers among the four (note they are selected at random for every IT-PIR query), the block position, pos, in the IT-PIR vector is set to one. For the the other two servers, the IT-PIR vector is exactly the same except for the block position pos which is now equal to 0. For a path $\mathcal{P}(\mathsf{tag})$, we denote blocks starting from the root to the leaf by $\{\mathcal{P}_1(\mathsf{tag}),\dots,\mathcal{P}_{\lambda\cdot L}(\mathsf{tag})\}$. The client retrieves from each server $\mathsf{rsl}_{i,j}$ for $i,j\in\{1,2\}$ (or $i,j\in[2]$ for simplicity), such that

$$\begin{aligned}
\bigoplus_{i,j\in[2]} \mathsf{rsl_{i,j}} &= \bigoplus_{\substack{k\in[\lambda\cdot L]\\ \mathsf{vect}_1[k]=1}} \left(\mathcal{P}_{1,k}(\mathsf{tag})\oplus\mathcal{P}_{2,k}(\mathsf{tag})\right)\\
&\quad \bigoplus_{\substack{m\in[\lambda\cdot L]\\ \mathsf{vect}_2[m]=1}} \left(\mathcal{P}_{3,m}(\mathsf{tag})\oplus\mathcal{P}_{4,m}(\mathsf{tag})\right)\\
&= \bigoplus_{\substack{k\in[\lambda\cdot L]\setminus\{\mathsf{pos}\}\\ \mathsf{vect}_1[k]=1}} \left(\mathcal{P}_{1,k}(\mathsf{tag})\oplus\mathcal{P}_{2,k}(\mathsf{tag})\right)\oplus\mathcal{P}_{1,\mathsf{pos}}(\mathsf{tag})\\
&\quad \bigoplus_{\substack{m\in[\lambda\cdot L]\setminus\{\mathsf{pos}\}\\ \mathsf{vect}_2[m]=1}} \left(\mathcal{P}_{3,m}(\mathsf{tag})\oplus\mathcal{P}_{4,m}(\mathsf{tag})\right)\oplus\mathcal{P}_{2,\mathsf{pos}}(\mathsf{tag})\\
&= \bigoplus_{\substack{k\in[\lambda\cdot L]\setminus\{\mathsf{pos}\}\\ \mathsf{vect}_1[k]=1}} \Big(\mathcal{P}_{1,k}(\mathsf{tag})\oplus\mathcal{P}_{2,k}(\mathsf{tag})\oplus\mathcal{P}_{1,k}(\mathsf{tag})\oplus\\
&\quad \mathcal{P}_{2,k}(\mathsf{tag})\Big)\oplus\mathcal{P}_{1,\mathsf{pos}}(\mathsf{tag})\oplus\mathcal{P}_{2,\mathsf{pos}}(\mathsf{tag})\\
&= \mathbf{0}\oplus P_{1,\mathsf{pos}}(\mathsf{tag})\oplus\mathcal{P}_{2,\mathsf{pos}}(\mathsf{tag})\\
&= \mathcal{P}_{1,\mathsf{pos}}(\mathsf{tag})\oplus\mathcal{P}_{2,\mathsf{pos}}(\mathsf{tag})=r\oplus r\oplus\mathsf{data}=\mathsf{data}
\end{aligned}$$

3.2.2 Merge

We show that the oblivious merge permutation, when applied to buckets in a specific path, preserves real blocks. First, from the C-ORAM correctness proof, we know that there is sufficient space in a child bucket to merge the parent's real elements.

So, the difficult part is to show that XORing blocks will not change the value of real and empty blocks. More formally, let $\mathcal{P}_j(\mathsf{tag},i) = (r^j_{i,1},\dots,r^j_{i,\lambda})$, $j\in[2]$ be a share of the i^{th} bucket on path $P(\mathsf{tag})$ containing λ blocks (shares). Consider two buckets stored in two servers containing the two shares $\mathcal{P}_j(\mathsf{tag},i)$ and $\mathcal{P}_j(\mathsf{tag},i+1)$, $j\in[2]$. Given an oblivious merge permutation π, we obtain

$$\mathcal{P}_j(\mathsf{tag},i)\oplus\pi(\mathcal{P}_j(\mathsf{tag},i+1))=(r^j_{i,1}\oplus r^j_{i+1,\pi(1)},\dots,r^j_{i,\lambda}\oplus r^j_{i+1,\pi(\lambda)}).$$

There are four different cases of XOR that can occur: an empty block with a real block, two empty blocks, two noisy blocks, and an empty block with a noisy block. We focus here on the first case, as the other three cases' correctness proof is similar. Given that a block data exists in the $(i+1)^{\text{th}}$ bucket at the k^{th} position, we need to show that $\bigoplus_{j\in[2]} r^j_{i,k}\oplus r^j_{i+1,\pi(k)}=\mathsf{data}$. Without loss of generality, assume that, before merging the buckets, the parent's block was the real block and the child's block was the empty block. Thus,

$$\begin{aligned}
\bigoplus_{j\in[2]}(r^j_{i,k}\oplus r^j_{i+1,\pi(k)}) &= r^1_{i,k}\oplus r^2_{i,k}\oplus r^1_{i+1,\pi(k)}\oplus r^2_{i+1,\pi(k)}\\
&= r^1_{i,k}\oplus r^1_{i,k}\oplus\mathsf{data}\oplus r^1_{i+1,\pi(k)}\oplus r^1_{i+1,\pi(k)}\\
&= \mathsf{data}.
\end{aligned}$$

Note that this holds as $r^2_{i,k}=r^1_{i,k}\oplus\mathsf{data}$. This is the result of our secret sharing technique.

3.2.3 Position Map: IT-Secure Path ORAM

To obliviously resolve a tag, the standard approach is to store the position map in a sequence of $\log N$ ORAMs of decreasing size [37]. As a result, either communication complexity of ORAM access is multiplied by a factor of $\log N$, or the minimum block size is increased by a factor of $\log N$.

It turns out that we can modify and then use Path ORAM [41] for the specific case of storing and accessing the position map in our setting. Path ORAM is a one-server construction that we transform to our multi-server no-encryption setting. Every tree in the recursive position map can be first replicated, and every block instead of being encrypted is secret shared in two other servers instead. To sum up, we have a structure similar to CH^f-ORAM in the sense that we store four versions of the position map on four servers.

3.3 CH^f-ORAM bandwidth analysis

In this section, we analyze CH^f-ORAM's communication complexity. For constant communication ORAM, we need to determine the lower bound of the block size that CH^f-ORAM can handle. In the following, the number of servers is constant, and therefore will be hidden in the big-O notation. Also, these results assume that the expansion factor ϕ is constant in the security parameter λ, the height L in $O(\log N)$ and $\lambda\in O(\log N)$. We will discuss the choice of parameters' values later in the security analysis, cf. Section 4.

ReadAndRemove: Assume that the client already knows the tag for the block they are looking for. With the 4 server setup, to read a block (Algorithm 2), the client: (1) downloads the three headers (size $O(L^2\cdot\lambda)$ bit as header_1 and header_2 have size equal to $\lambda\cdot L$ per bucket, and header_1 has a size equal to λ per bucket), (2) sends the IT-PIR query (size $O(\lambda\cdot L)$ bit as it consists of a sequence of bits equal to the number of blocks in a path), and (3) downloads the XORed output (size $O(B)$ bit as the client download one block from each server). This computes to a total of $O(L^2\cdot\lambda+\lambda\cdot L+B)=O(\log^3 N+B)$.

Add: The clients uploads ϕ blocks (size $O(B)$ bit) back to the root bucket. That is, the communication complexity is in $O(\phi\cdot B)$. As ϕ is constant, the add communication complexity equals to $O(B)$

Evict: To evict a path, the client needs to download all headers, generate permutations for any two adjacent buckets based on oblivious merge algorithm. The number of required bit for the eviction is: (1) $O(\lambda\cdot L^2)$ bits required for the headers, (2) $O(L\cdot\lambda\cdot\log\lambda)$ bits required for the permutation as one permutation has a size equal to $\lambda\cdot\log\lambda$ bits. That is, the overall communication complexity equals $O(\lambda\cdot L^2+L\cdot\lambda\cdot\log\lambda)=O(\log^3 N)$. The eviction is a process that occurs after $\chi=O(\log N)$ read operations. That is, the amortized number of bit transferred between the client and servers equals $O(\log^2 N)$.

Position Map: The position map access consists of accessing recursively a logarithmic number of Path ORAM trees. Recall that to access one Path ORAM tree in the position map, $O(\log^2 N)$ bits are required as the block size equals to $\log N$ and the bucket size has a constant number of blocks equal to 5 (or 4 empirically). The overall position map communication complexity then equals $O(\log^3 N)$.

Overall communication complexity: The entire communication complexity required to fetch a block while taking into consideration the position map communication complexity, is equal to

$$O(\underbrace{\log^3 N+B+\log^2 N}_{\text{ReadAndRemove, Add and eviction}}+\underbrace{\log N\cdot(\log^2 N)}_{\text{Position map access}}).$$

To achieve $O(1)$ communication complexity, we set block size $B \in \Omega(\log^3 N)$. That is, CH^f-ORAM enjoys a block size that is smaller than related work by a factor of $\log N$.

Client Storage: Path ORAM position maps requires a stash for every tree. The stash's size is in $O(\log N \cdot B_p)$, where $B_p = O(\log N)$ is the block size in the position map. That is, Path ORAM position map requires a client memory in $O(\log^3 N)$, taking into account the $\log N$ recursive trees. Fortunately, choosing a block size in $\Omega(\log^3 N)$, makes the entire client storage for the stash equal to one block of CH^f-ORAM. That is, we do not loose the constant client storage property of CH^f-ORAM.

3.4 Integrity

Ren et al. [34] show that a malicious server can violate an ORAM's security by sending back specific sequences of wrong data in response to a client's request. As a consequence, ORAMs need an integrity protection mechanism that detects whether a server tampered with returned data. While for Path ORAM a simple Merkle tree-based approach is sufficient, recent PIR-ORAM hybrids require more sophisticated solutions for integrity. For example, see the coding based approach in Onion ORAM [11].

While one could adopt Onion ORAM's mechanisms, integrity for CH^f-ORAM turns out to be straightforward thanks to using XOR-based IT-PIR. So, we only outline the main concepts below.

So far, a block in CH^f-ORAM is a simple bit string containing data. For integrity protection, instead of only storing data, we now store in a block the concatenation $\mathsf{data}||\mathsf{HMAC}_K(\mathsf{data})$ with a key K only known to the client. As before, each server computes an XOR over all blocks (each containing a hash now) and sends the resulting bit string back to the client. The client then XORs all received bit strings and verifies whether the HMAC of this final bit string (the last λ bit) matches the data part (the first B bit). If they do not match, the client has detected malicious behavior and stops.

Security follows, as servers do not collude: they do neither know the other servers' state nor can they coordinate in what to return to the client. Each bit that a server changes in the reply will break HMAC verification.

4. SECURITY ANALYSIS

Security definition: Our security definition captures the standard ORAM security intuition that, as long as there are no overflows in the ORAM tree (or in the position map Path ORAM stashes), two access patterns are computationally indistinguishable.

DEFINITION 4.1. *Let λ be the security parameter. Let $\overrightarrow{a} = \{(op_1,d_1,a_1),(op_2,d_2,a_2),\dots,(op_M,d_M,a_M)\}$ be a sequence accesses (op_i,d_i,a_i), where op_i denotes a* ReadAndRemove *or an* Add *operation, a_i the address of the block, and d_i the data to be written if $op_i =$ Add, or $d_i = \perp$ when $op_i =$ ReadAndRemove. Let* Overf *be the event that a bucket overflows during an access.*

Let $A(\overrightarrow{a},x)$ be all communications sent between the client and server x in executing the access pattern $\overrightarrow{a}$.

We say that CH^f-ORAM is secure iff *for all probabilistic polynomial-time (in λ) adversaries $\mathcal{D}$ there exists a negligible function* negl *such that for sufficiently large λ*

1. *the probability of overflow $\Pr(\mathsf{Overf}) \le \mathsf{negl}(\lambda)$ and*

2. *for all values of x and all sequences $\overrightarrow{a}$ and $\overrightarrow{b}$, where $|\overrightarrow{a}| = |\overrightarrow{b}| \in poly(\lambda)$*

$$|Pr[\mathcal{D}(A(\overrightarrow{a},x)) = 1] - Pr[\mathcal{D}(A(\overrightarrow{b},x)) = 1]| \le \mathsf{negl}(\lambda).$$

4.1 Overflow Analysis

Our eviction is similar to various previous works [11, 28, 35]. First, the eviction path is selected following a reverse lexicographic order. After every read operation, the element that was read is put back in the root. After χ read operations an eviction occurs. To achieve the same overflow probability as previous work, the size of the bucket can be set to those of previous works. Along the same lines, eviction rate and the tree height are as before. Consequently, we only repeat the main results and refer the reader to Devadas et al. [11] for details and proofs.

THEOREM 2. *For eviction rate χ and tree height L, with $\lambda \ge \chi$ and $N \le \chi \cdot 2^{L-1}$, the probability that a bucket overflows is upper bounded by $e^{\frac{-(2\lambda-\chi)^2}{6\cdot\chi}}$.*

For an overflow probability negligible in the security parameter λ, it is sufficient to choose $L \in \Theta(\log N)$, $\chi \in \Theta(\lambda)$. In practice, we can choose $\lambda \in O(\log N)$.

Additionally, in CH^f-ORAM, the number of empty blocks in all buckets have to be sufficient to handle noisy blocks. Noisy blocks are inherent to the oblivious merge technique. Since our eviction and oblivious merge process are similar to those in C-ORAM, we borrow their theorem result and refer the reader to [28] for proof details.

THEOREM 3. *If $\phi \in \Theta(1)$, a real block gets overwritten with a probability in $O(\lambda^{-\lambda})$.*

If bucket size $\lambda \in O(\log N)$, $L \in \Theta(\log N)$, and $\phi \in \Theta(1)$, the probability that a real block gets overwritten is in $O(N^{-\log\log N})$. Experiments [28] show that empirically $\phi \approx 2.2$ is sufficient.

ϕ is an expansion factor that needs to be adapted to our batch eviction trick. In C-ORAM, the eviction was performed after every access. This made the bucket size smaller for reasonable overflow probability, and in particular, the eviction rate was not necessary. Now, with batch eviction, we evict less often with more real blocks in the root bucket. Based on results of [28], the probability that a real block gets overwritten at the i^{th} level and j^{th} eviction, $\Pr[R_{i,j}]$, equals $\Pr[R_{i,j}] \le Me^{i+\ln(i\cdot\chi)+4\phi\cdot\chi-\ln(\phi\cdot\chi)\cdot\phi\cdot\chi}$, where M is a constant. We can upper-bound the above quantity for all $i \in [L]$ and for all $j \in \mathbb{N}$ such that

$$\Pr[R] \le Me^{L+\ln(L\cdot\chi)+4\phi\cdot\chi-\ln(\phi\cdot\chi)\cdot\phi\cdot\chi}.$$

Let e^λ be the number of operations that we want to perform on CH^f-ORAM before an overflow occurs. That is, with a simple union bound we have

$$\begin{aligned}\Pr[\bigcup_{\substack{i\in[L]\\ j\in[e^\lambda]}} R] &\le \sum_{\substack{i\in[L]\\ j\in[e^\lambda]}} \Pr[R]\\ &\le Me^{z+\ln L+L+\ln(L\cdot\chi)+4\phi\cdot\chi-\ln(\phi\cdot\chi)\cdot\phi\cdot\chi},\end{aligned}$$

For example, the bucket size should be equal to $\chi \cdot \phi = 140$ to have an overflow of $\approx 2^{-20}$, for an $L = 30$.

4.2 Oblivious merge with batch insertion

Batch insertion consists of adding, after every access, ϕ blocks to the root. Among these ϕ blocks, $\phi - 1$ are empty and one block is real. These ϕ blocks are randomly shuffled before being inserted to the root. That is, after χ accesses, the bucket can be seen as being composed of χ sub-buckets that each contains ϕ blocks. We formally define a sub-bucket below.

DEFINITION 4.2. *For all $\mathsf{tag} \in \{0,1\}^L$, $i \le L$, given a bucket $\mathcal{P}(\mathsf{tag},i)$ composed of λ blocks such that $\lambda = \phi \cdot \chi$, the j^{th} sub-bucket of $\mathcal{P}(\mathsf{tag},i)$ consists of the sequence of ϕ blocks at position*

$\{(j-1)\cdot\phi+1,\cdots,\phi\cdot j\}$, *for all* $1\leq j\leq\chi$. *We denote by* $\mathcal{P}^j(\mathsf{tag},i)$ *the* j^{th} *sub-bucket of* $\mathcal{P}(\mathsf{tag},i)$.

Public leakage of batch insertion: First notice that the server learns that, at the root bucket, real blocks are distributed in the χ sub-buckets of the root. In other words, the server knows that every sub-bucket of $\mathcal{P}^j(0,0)$, for all $j\leq\chi$, exactly contains *one* real block. On the other hand, the batch insertion does not only have an impact on the distribution of blocks in the root bucket, but also on other buckets of the tree. As an instance, the sibling of the root is an exact copy of the root, based on our eviction process. That is, the root sibling has the same distribution as the root bucket. In general, batch insertion impacts all buckets of the ORAM tree in such a way that all buckets will preserve the root initial distribution at some differences that we detail below.

All real blocks that were inserted in the root can eventually turn to noisy blocks in lower levels of the tree. That is, the batch insertion will imply that for every sub-bucket $\mathcal{P}^j(\mathsf{tag},i)$, for all $\mathsf{tag}\in\{0,1\}^L$, $i\leq L$ and $j\leq\chi$, there is *at least* one real or noisy block in every sub-bucket.

A valid question to ask is if such distribution would give more chance to the server to determine the permutation generated by the oblivious merge algorithm. Recall that in C-ORAM, the bucket distribution is *unknown* to the server and the permutation outputted by the oblivious merge algorithm is indistinguishable from a random permutation.

Having a public knowledge of blocks' distribution would imply ultimately that the client cannot generate all permutations from $\{\{1,\cdots,\lambda\}\rightarrow\{1,\cdots,\lambda\}\}$. The server also knows that the client cannot generate all possible permutations. Thus, the server is not anymore distinguishing against a randomly permutation π generated from $\{\{1,\cdots,\lambda\}\rightarrow\{1,\cdots,\lambda\}\}$, but against a new ideal restrained permutation that takes into account the new a-priori public knowledge of the server. We call this new permutation *batch permutation* that we are going to define formally below. Note that this is a public knowledge that the adversary will use in order to eliminate many possible permutations. Before defining *batch permutation*, we first define the batch insertion event.

DEFINITION 4.3. *For all* $\phi,\chi>0$, *a* batch insertion event, $\mathsf{AI}(\phi,\chi)$, *is the event of inserting at random* χ *real or noisy blocks each in the* j^{th} *sub-bucket* $\mathcal{P}^j(\mathsf{tag},i)$, *for all* $\mathsf{tag}\in\{0,1\}^L$, $i\leq L$ *and* $j\leq\chi$.

LEMMA 4.1. *A* batch permutation *is a random permutation constrained to the event* $\mathsf{AI}_{\phi,\chi}$ *such that for all* $\phi,\chi>0$ *s.t.* $\lambda=\phi\cdot\chi$

$$\Pr[\pi\leftarrow\{\{1,\cdots,\lambda\}\rightarrow\{1,\cdots,\lambda\}\}\mid\mathsf{AI}_{\phi,\chi}]=\frac{1}{\binom{\phi}{1}^{\chi}\cdot\chi!\cdot(\lambda-\chi)!}$$

We denote $\mathsf{AP}_{\phi,\chi}$ the set of all possible permutations for $\phi,\chi>0$. Note that it is clear that $|\mathsf{APerm}_{\phi,\chi}|\leq|\mathsf{Perm}|$, where $\mathsf{Perm}=\{\{1,\cdots,\lambda\}\rightarrow\{1,\cdots,\lambda\}\}$, for $\lambda=\phi\cdot\chi$. It is important to show that the oblivious merge algorithm, given the batch insertion, outputs permutations that are indistinguishable from batch permutations for computationally unbounded adversaries. That is, given a batch permutation, the adversary does not get any additional information about the load of the bucket. In particular, quantifying the bucket's load and the permutation output are two independent events.

Adversarial model: Let $\mathsf{GenPerm}$ be the (probabilistic) oblivious merge algorithm. Our adversarial model is the following: an adversary can chose any pair of two adjacent buckets' headers, and can only get one permutation output by $\mathsf{GenPerm}$ for it. The header follows the batch insertion bucket distribution from an adversarial view, described above. This operation can be performed an unbounded number of times under the condition that buckets headers have to be updated following an oblivious merge operation, or different from any previous adversarial request.

Batch insertion main Theorem: Let $\mathsf{Load}\big((i,n_A),(j,n_B)\big)$ be the event that the number of real and noisy blocks in A and B are respectively i, j, and n_A and n_B. k represents the size of the buckets such that $\lambda=\phi\cdot\chi$, for all $\phi,\chi>0$.

THEOREM 4. *Let* H_A *and* H_B *be the respective headers of two buckets A and B such that for all permutations* π' *from* $\mathsf{APerm}_{\phi,\chi}$,

$$\Pr[\pi\leftarrow\mathsf{GenPerm}(\mathsf{H}_A,\mathsf{H}_B)=\pi'\mid\mathsf{AP}_{\phi,\chi}]=\frac{1}{\binom{\phi}{1}^{\chi}\cdot\chi!\cdot(\lambda-\chi)!}.$$

PROOF. We prove the theorem for any two adjacent buckets A and B. For the case of A being the root, it would be sufficient to set n_A to 0 in the proof. Throughout the proof, a *combination* of a bucket refers to the distribution of real, empty and noisy blocks. From a fixed combination, we can generate all possible permutations by taking into consideration the blocks' positions.

Let Π be the random variable that captures which permutation has been outputted by $\mathsf{GenPerm}$. For every possible combination of real, empty, and noisy blocks in B, there are only few possible combinations for B that can be selected, for correctness reasons. These combinations are called valid combinations of B, and their set is denoted by VC. Let C_B be the random variable that captures the combination that bucket B has before being merged with A. Given that bucket B has j real blocks and n_B noisy blocks, the overall number of possible combinations while taking into account the event $\mathsf{AI}_{\phi,\chi}$ equals $\underbrace{\binom{\phi}{1}\cdots\binom{\phi}{1}}_{\chi\text{ times}}\cdot\binom{\lambda-\chi}{n_B+j-\chi}\cdot\binom{n_B+j-\chi}{j-\chi}$. To understand the counting process behind this formula, first note that χ blocks have to be in each of the χ-subsets of the bucket based on $\mathsf{AI}_{\phi,\chi}$ event so the ϕ^{χ} term, then the remaining real and noisy blocks have to be selected so the $\binom{\lambda-\chi}{n_B+j-\chi}$ term and finally as the real and noisy block are two different entities we need also to count possible cominations among the prior selected. That is, for any fixed possible combinations δ_B of bucket B

Let Π be the random variable that captures which permutation has been outputted by $\mathsf{GenPerm}$. For every possible combination of real, empty, and noisy blocks in B, there are only few possible combinations for B that can be selected, for correctness reasons. These combinations are called valid combinations of B, and their set is denoted by VC. Let C_B be the random variable that captures the combination that bucket B has before being merged with A. Given that bucket B has j real blocks and n_B noisy blocks, the overall number of possible combinations while taking into account the event $\mathsf{AI}_{\phi,\chi}$ equals $\underbrace{\binom{\phi}{1}\cdots\binom{\phi}{1}}_{\chi times}\cdot\binom{\lambda-\chi}{n_B+j-\chi}\cdot\binom{n_B+j-\chi}{j-\chi}$. To understand the counting process behind this formula, first note that χ blocks have to be in each of the χ-subsets of the bucket based on $\mathsf{AI}_{\phi,\chi}$ event so the ϕ^{χ} term, then the remaining real and noisy blocks have to be selected so the $\binom{\lambda-\chi}{n_B+j-\chi}$ term and finally as the real and noisy block are two different entities we need also to count possible combinations among the prior selected. That is, for any fixed possible combinations δ_B of bucket B

$$\Pr[C_B=\delta_B]=\frac{1}{\phi^{\chi}\cdot\binom{\lambda-\chi}{n_B+j-\chi}\cdot\binom{n_B+j-\chi}{j-\chi}}.$$

Besides, the probability that random variable Π is equal to a specific permutation π', conditionally to the event $\mathsf{AI}_{\phi,\chi}$, computes to

$$\begin{aligned}
\Pr[\Pi=\pi'|\mathsf{AI}_{\phi,\chi}] &= \sum_{i,j\in[\lambda]} \Pr[\Pi=\pi' \wedge \mathsf{Load}\big((i,n_A),(j,n_B)\big)|\mathsf{AI}_{\phi,\chi}] \\
&= \sum_{i,j\in[\lambda]} \Pr[\Pi=\pi' \mid \mathsf{Load}\big((i,n_A),(j,n_B)\big)\wedge\mathsf{AI}_{\phi,\chi}]\cdot \\
&\quad \Pr[\mathsf{Load}\big((i,n_A),(j,n_B)\big)] \\
&= \sum_{i,j\in[\lambda]}\sum_{\delta_B\in\mathsf{VC}} \Pr[\mathsf{Load}\big((i,n_A),(j,n_B)\big)]\cdot \\
&\quad \Pr[\Pi=\pi'|\mathsf{Load}\big((i,n_A),(j,n_B)\big)\wedge\mathsf{AI}_{\phi,\chi}\wedge C_B=\delta_B]\cdot \\
&\quad \Pr[C_B=\delta_B]
\end{aligned}$$

Now, we need to quantify the probability of the event $E=\{\Pi=\pi'|\mathsf{Load}\big((i,n_A),(j,n_B)\big),\mathsf{AI}_{\phi,\chi},\, C=\delta_B\}$ as well as the cardinality of the set of valid combinations VC. We present these two results in the following two lemmas.

LEMMA 4.2. *For all $\chi,\phi>0$, for all permutations π from $\mathsf{APerm}_{\phi,\chi}$, for all $n_A,n_B,i,j\in[\lambda]$ such that $n_A+i<\lambda$ and $n_B+j<\lambda$*

$$\Pr[E]=\frac{1}{n_A!\chi!\cdot(i-\chi)!\cdot(\lambda-n_A-i)!\cdot\phi^{\chi}\binom{\lambda-j-n_B}{i+n_A-\chi}\cdot\binom{i+n_A-\chi}{i-\chi}}.$$

PROOF. First, note that the event E captures the following: the probability that a permutation is generated for a fixed combination of B and load of A and B. That is, we proceed by counting all possible such permutations. Given a valid combination δ_B for a fixed load of A and B, we need to calculate the number of valid combinations of A for this a-priori fixed combination of B, δ_B. This quantity equals $\phi^{\chi}\binom{\lambda-j-n_B}{i+n_A-\chi}\cdot\binom{i+n_A-\chi}{i-\chi}$. The first term of this quantity computes the number of combinations to insert χ real blocks into the χ-subsets of the bucket, the second term underlines the following: in order to merge correctly the buckets A and B, the real and noisy elements needs to be added to empty blocks. That is we have to insert $i+n_A-\chi$ into $\lambda-j-n_B$ empty blocks in B. The last term is only to quantify the number of combinations in $i+n_A-\chi$ as the noisy and real blocks are distinct. Given the number of combinations, we can easily find the valid number of permutations that can be applied over A. We need to permute four different entities among each others: (1) the χ inserted real (or all noisy) blocks, (2) the $(i-\chi)$ remaining real blocks, (3) n_A noisy blocks, and (4) $\lambda-n_A-i$ remaining empty blocks such that: $n_A!\cdot i!\cdot(\lambda-n_A-i)!\cdot\binom{\lambda-j-n_B}{i}\cdot\binom{\lambda-j-i}{n_A}$. Thus the lemma result. □

An important remark at this stage would be to notice that the probability of event E and the probability to select a combination in B, $\Pr[C_B=\delta_B]$, are both independent of the combination δ_B. This implies that

$$\sum_{\delta_B\in\mathsf{VC}} \Pr[E]\cdot\Pr[C=\delta_B]=|\mathsf{VC}|\cdot\Pr[E]\cdot\Pr[C=\delta_B]$$

Now, given a fixed combination over bucket A, we need to find out the number of *valid* combinations of B, $|\mathsf{VC}|$. We present it under the form of a lemma:

LEMMA 4.3. *For all $\chi,\phi>0$, for all $n_A,n_B,i,j\in[k]$*

$$|\mathsf{VC}|=\phi^{\chi}\begin{pmatrix}\lambda-\chi\\ n_B+j-\chi\end{pmatrix}\cdot\begin{pmatrix}n_b+j-\chi\\ j-\chi\end{pmatrix}.$$

PROOF. This is a counting problem that can be solved as follows: note first that a combination over A is valid iff all real elements in both buckets A and B were not overwritten by any noisy block. That is, for a fixed combination over A, we need to enumerate the number of combinations over B that satisfy this statement. A valid combination of B is therefore the one for which real elements in B have empty blocks in the combination of A. This computes to $\phi^{\chi}\cdot\binom{\lambda-i-n_A}{n_B+j-\chi}\cdot\binom{n_B+j-\chi}{j-\chi}$. The first term, similar to the previous counting argument, captures the event $\mathsf{AI}_{\phi,\chi}$. The second term illustrates the possible combinations of remaining blocks. The last term determines the number of combinations of noisy and real blocks per possible combination. □

Putting all together, we can verify that

$$|\mathsf{VC}|\cdot\Pr[E]\cdot\Pr[C=\delta_B]=\frac{1}{\phi^{\chi}\cdot\chi!\cdot(\lambda-\chi)!}.$$

Therefore,

$$\begin{aligned}
\Pr[\Pi=\pi' \mid \mathsf{AP}_{\phi,\chi}] &= \sum_{i,j\in[\lambda]} |\mathsf{VC}|\cdot\Pr[E]\cdot\Pr[C=\delta_B] \\
&\quad \cdot\Pr[\mathsf{Load}\big((i,n_A),(j,n_B)\big)] \\
&= \sum_{i,j\in[\lambda]} \frac{1}{\phi^{\chi}\cdot\chi!\cdot(\lambda-\chi)!}\cdot\Pr[\mathsf{Load}\big((i,n_A),(j,n_B)\big) \\
&= \frac{1}{\phi^{\chi}\cdot\chi!\cdot(\lambda-\chi)!}
\end{aligned}$$

□

5. EVALUATION

In order to compare CH^f-ORAM to related work, we have to derive concrete values for some of the parameters which were only expressed asymptotically above. In particular, although λ is $O(\log N)$ and $\chi=\Theta(\lambda)$, exact values are needed for an implementation.

For our schemes to have good communication complexity, χ should be as large as possible. However, the larger χ is the higher the probability of a bucket overflow during eviction. The ratio between χ and λ is ϕ and represents the communication cost for a query. Every query must write ϕ blocks to the root node. Figure 2 shows χ versus the number of operations that an instance of CH^f-ORAM is able to support before an overflow, as determined experimentally (note the log scale). As χ increases, the number of operations before an overflow drops off dramatically. For our experiments, we chose $\chi=\lambda/10$

For determining λ, we created multiple instances of CH^f-ORAM with $N=2^{15}$ and various settings of λ, and then executed accesses on them until an overflow occurred. Taking the average over 20 runs for each value of λ, we obtained Figure 3. Extrapolating, we can see that for 50 bits of security, CH^f-ORAM requires λ to be approximately 900. This is substantially larger than related constant-communication schemes like C-ORAM [28]. However, the bucket size has very little impact on the cost of CH^f-ORAM. The only communication that is performed over buckets is the headers, which amount to only a few thousand bits even with large λ.

The more important consideration with λ being so large is how it impacts server computation time, since the servers must compute over all buckets in a path to perform PIR and eviction. Figure 4 shows overall query execution time, including network transfer and server computation for CH^f-ORAM and the current state of the art scheme Ring ORAM [35] with various values of N and a block size of 1 MB. We assume a network speed of 20 Mbps. Furthermore, we assume that each server is equally powerful and can calculate the XOR of two 1 MB blocks in 1 ms (the amount of time it took on our test machine, a 2012 Macbook Pro with 2.4 Ghz Intel i7 processor). Note that our evaluation focuses on communication and computation overhead as being the main metrics on which ORAM

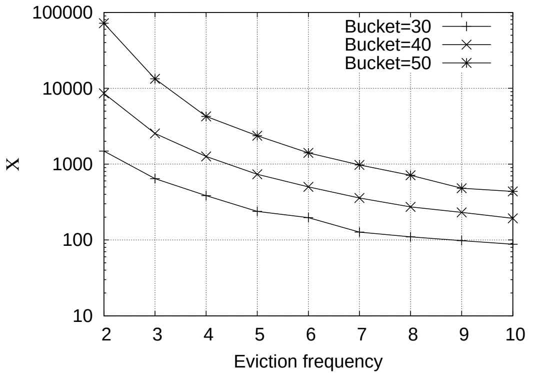

Figure 2: Eviction rate χ

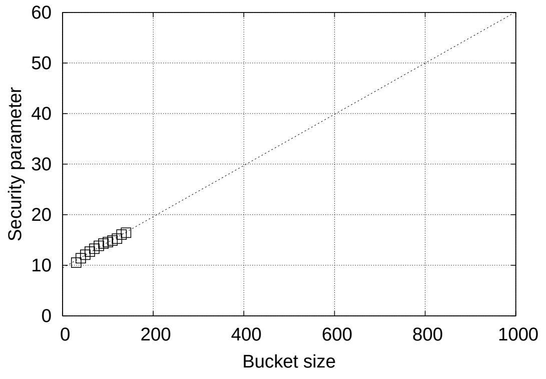

Figure 3: Bucket size vs security parameter

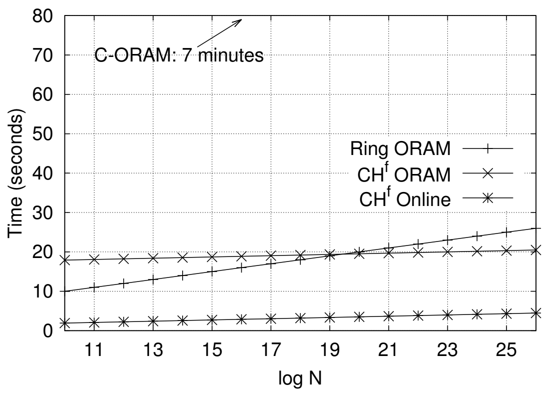

Figure 4: $\log N$ vs time to perform a query

constructions are compared to. As stated earlier, we did not modify Ring ORAM [35] single-server setting. Increasing the number of servers while preserving a non-communicating servers setting does not have any trivial positive impact on Ring ORAM construction communication overhead.

The results show that CH^f-ORAM beats Ring ORAM, starting from $N \sim 2^{20}$. The cost of CH^f-ORAM is dominated by the ϕ blocks that must be uploaded for every access. To 4 servers, with $\chi = \lambda/10$, this amounts to 40 MB. Beyond that, the server computation is almost negligible which is why CH^f-ORAM scales much better for larger N. Note that CH^f-ORAM offers constant memory complexity on the client, while Ring ORAM requires $O(\log N)$. Compared to schemes with homomorphic encryption, like Onion or C-ORAM, a query in our ORAM can be performed in less than a second rather than several minutes. A significant advantages especially for resource-constrained devices. On the other hand note that the *online* communication cost of CH^f-ORAM is below 5 seconds for $N \sim 2^{20}$, which shows that our construction is extremely suitable for fast read operations—recall that online cost captures the time required to access the block while not taking into consideration eviction process.

Also interesting is the increased efficiency on the client side for CH^f-ORAM. The client performs no encryption operations, only XOR on four blocks and generating ϕ blocks of random data. In contrast, Ring ORAM clients perform decryption and reencryption on $\sim 2.5\log N$ blocks. This encryption time is not accounted for in Figure 4 because it highly depends on the power of the client, but could easily double the overall execution time on a low powered device. This makes CH^f-ORAM uniquely attractive to devices with low computational ability.

6. SECURE RAM COMPUTATION

ORAM has been recently extensively used for secure RAM computation [21, 24, 25, 42]. In the following, we give a high level description on how CH^f-ORAM can be utilized as a component for secure RAM computation. For further details, we refer the reader to [21, 30]. We picture the four servers of CH^f-ORAM as four parties that desire to compute a RAM program on specific data. This data can be composed of parties input or, as demonstrated by Gordon et al. [21], be a-priori stored in a secure multi-party computation fashion. The secure RAM computation requires that the client's state, as well the memory, be secret shared between the parties. Besides the $T-$time RAM program, there is also a sequence of T instructions that are evaluated via secure computation that takes as inputs the secret shares of the ORAM memory, the client and the program state. As defined in previous works, both the program logic and memory operations are implemented based on secure computation. It has been pointed out in SCORAM [43] that if a tree-based ORAM is the underlying ORAM, the eviction process when modeled as a circuit can add a non-negligible overhead.

One of the interesting features of CH^f-ORAM is that its eviction is independent of the block size. This stems from the fact that CH^f-ORAM's eviction only requires transferring the buckets' headers on a specific path, plus the oblivious merge permutations, which are both independent of the block size.

An additional feature that is important for an ORAM to work well in a secure RAM setting is that its client state be small. As shown by Lu and Ostrovsky [25], the state must be exchanged between the clients during every step of the computation. Therefore, reducing the size of the state is extremely important for secure RAM computation as it will reduce the circuit size. CH^f-ORAM has $O(1)$ client memory, inducing the lowest possible overhead on the secure RAM construction. In contrast with existing work, all sublinear ORAM inducing $O(\sqrt{N})$ or $O(\log N)$ client storage such as [40, 44, 45] would be very inefficient in this secure RAM setting.

From table 2, it is clear that CH^f-ORAM offers a constant size circuit if $B \in \widetilde{\Omega}(\lambda \cdot \log^2 N)$, where λ is the security parameter. As far as we know, this is best asymptotic size so far in literature for

Table 2: Circuit size of eviction process for recent ORAM constructions. λ, N , C_{PRF} and B respectively denote the security parameter, the number of blocks, the circuit size of a PRF, and the block size. $\widetilde{O}$ hide a $\log(\lambda)$ factor.

Scheme	Circuit Size (Asymptotics)
Shi et al. [37]	$O\big((\log^3 N + B\cdot\log N)\cdot\lambda\big)$
Chung et al. [7]	$O\big((\log^3 N + B\cdot\log N)\cdot\lambda\big)$
Path SC ORAM [43]	$\widetilde{O}\big((\log^2 N + B)\cdot\lambda\big)$
LO [25]	$O\big((C_{PRF} + B)\cdot\log N\big)$
Circuit ORAM [42]	$O\big((\log^2 N + B)\cdot\lambda\big)$
CH^f-ORAM	$O(\log\lambda\cdot\lambda\cdot\log^2 N + B)$

the eviction process as it is independent of the block size. It even beats the well-known lower bound of Goldreich and Ostrovsky as it applies to the circuit size metric [42]. This is possible because the lower bound only captures settings where the server does no perform computation. Since we have the server do XOR operations to simplify eviction from the client side, it circumvents that lower bound and allows us to achieve $O(1)$ complexity.

Previously, Circuit ORAM [42] was the construction that provided the best circuit size asymptotics, as it approaches the Goldreich and Ostrovsky [17] lower bound for block sizes in $\Omega(\log^2 N)$. Wang et al. [43] presented SCORAM an adapted tree-based ORAM for secure RAM computation. Authors did not provide asymptotics about their circuit size, as it is based only on heuristics. As noted in [43], asymptotics might hide larger constants as it is the case for Path SC ORAM, the adaptation of Path ORAM for secure computation, where the logarithmic client state is stored in the main memory and oblivious sorting is used instead in order to decrease the state size (from $O(\log N)\cdot\omega(1)$ to $O(1)$). As oblivious sorting hides larger constants, in Path SC ORAM, the asymptotics do not reflect the practical behavior.

Finally, having constant circuit size in B leads to the following general result: if a RAM program Π has a running time equal to $O(T)$ and under the existence of a constant round multy-party computation, then every instruction will require $O(1)$ communication overhead, and the RAM program can be securely computed in $O(T)$—Note that the big-O is in the block size that we assume is in $\widetilde{\Omega}(\lambda\cdot\log^3 N)$.

7. RELATED WORK

Oblivious RAM goes back to the seminal paper by Goldreich and Ostrovsky [17]. There have been several attempts to improve different aspects of ORAM, such as its communication complexity, number of interactions between the server and the client, memory complexity on the client side, and storage and computation overhead on the server [6, 10, 15–20, 23, 26–30, 32, 35, 37, 41, 44, 45]. We briefly review three ORAM categorizations. The first discusses recent advances of schemes with constant client memory complexity, the second targets schemes with sublinear client memory, and the third presents recent works in multiple-servers ORAM.

Constant client memory: Constant client memory is very appealing for resource-constrained devices with limited memory, e.g., embedded devices, small sensors, and devices in the Internet of Things. Moreover, constant client memory is very useful in trusted proxy settings when used by one or more clients, so there is no need to transfer a large state. Goodrich and Mitzenmacher [18] and Pinkas and Reinman [32] introduced amortize communication complexity in $O(\log^2 N)$, but with linear worst-case communication complexity. Shi et al. [37] introduce tree-based structures providing a worst-case poly-logarithmic communication complexity in $O(\log^3 N)$ blocks. Many subsequent papers build on top of this one to further decrease communication or storage complexity storage [15, 26, 27, 29]. Recently, there have been many attempts to decrease the communication overhead to be constant in the number of blocks. That is, obliviously reading or writing a block with only a constant number of transferred blocks as overhead. Using servers with computational capabilities instead of storage-only servers, Devadas et al. [11] showed how to construct a constant communication ORAM for blocks in $\Omega(\log^5 N)$. Fletcher et al. [13] show how to decrease the number of interactions of Onion ORAM from $\log N$ to 1. Moataz et al. [28] demonstrate how to preserve constant communication for smaller block size in $\Omega(\log^4 N)$, while performing eviction with fewer number of homomorphic multiplications. Although low asymptotic bounds have been reached for communication complexity, high computational latency on server side makes constant client memory not yet ready for deployment [28].

Poly-log client memory: Earlier schemes have memory complexity on the client side in $O(\sqrt{N})$, yet inducing a linear worst-case communication complexity [44, 45]. Stefanov et al. [40] show how to get a worst-case memory complexity in $O(\sqrt{N})$ with a a communication complexity in $O(\log^2 N)$. Stefanov et al. [41] present how to provide a $O(\log N)$ communication complexity with only a logarithmic memory complexity on the client size. This scheme has been improved by multiplicative constant in [27, 35]. Recently, Garg et al. [14] improves the number of interactions of Path ORAM to be constant while inducing a multiplicative security overhead factor.

Distributed setting: Many ORAMs leverage multiple servers to decrease overhead. For example, ObliviStore [39] decreases overhead using an oblivious load balancing technique relying on *trusted* internal nodes to distribute accesses. Stefanov and Shi [38] accesses blocks in $O(1)$ in a two-servers setting (extendable to k servers) with $O(\log N)$ communication complexity between servers and $O(\sqrt{N})$ client storage complexity. Lu and Ostrovsky [25] show how to achieve Goldreich and Ostrovsky's lower bound $O(\log N)$ with two non-communicating servers and with $O(1)$ client storage complexity. Dachman-Soled et al. [9] introduce the notion of oblivious network RAM that can be also fit to a distributed setting and decreases access complexity to $O(1)$. However this comes at the cost of a weaker security model where the adversary is only allowed to observe communication between servers and client. Servers themselves are trusted to not reveal details about queries.

In conclusion, all distributed ORAM's today fail to offer constant communication overhead with no concession, either by weakening the threat model or by leveraging communication between the servers.

Moreover, due to the use of encryption, the security of related work is based on a computational hardness assumption. In contrast, Damgård et al. [10] and Ajtai [1] propose ORAMs without computational hardness assumptions. Tree-based ORAMs with a storage-only server can easily get rid of encryption, leveraging two non-communicating servers, and be converted to an information-theoretic ORAM. This paper tackles information-theoretic security in a storage-computing server. Information-theoretic security is especially interesting in situations where encryption would overburden devices' computational capabilities, e.g., wireless sensors.

8. CONCLUSION

CH^f-ORAM is a new constant communication complexity, constant client memory complexity ORAM that avoids expensive homomorphic encryption. This makes it especially attractive in scenarios with resource-constrained client devices. Our evaluation demonstrates that CH^f-ORAM is up to two orders of magnitudes faster than related work such as C ORAM. Towards practicality, another advantage is a block size smaller by a multiplicative factor of $\log N$ compared to other homomorphic encryption-based solutions. CH^f-ORAM's novel eviction circuit has size constant in the ORAM's block size. This makes it attractive to apply within recently introduced Secure RAM Computations.

References

[1] Miklós Ajtai. Oblivious rams without cryptogrpahic assumptions. In *Proceedings of Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 181–190, 2010.

[2] A. Ambainis. Upper bound on communication complexity of private information retrieval. In *International Colloquium on Automata, Languages and Programming*, pages 401–407, Bologna, Italy, 1997.

[3] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond. Breaking the $O(n^{1/(2k-1)})$ Barrier for Information-Theoretic Private Information Retrieval. In *Symposium on Foundations of Computer Science*, pages 261–270, Vancouver, Canada, 2002.

[4] J. Benaloh and J. Leichter. Generalized secret sharing and monotone functions. In *Proceedings on Advances in cryptology*, pages 27–35, 1990.

[5] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Symposium on Foundations of Computer Science*, pages 41–50, Milwaukee, USA, 1995.

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

[7] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with õ(log^2 n) overhead. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 62–81, 2014.

[8] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.

[9] D. Dachman-Soled, C. Liu, C. Papamanthou, E. Shi, and U. Vishkin. Oblivious network RAM and leveraging parallelism to achieve obliviousness. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, pages 337–359, 2015.

[10] I. Damgård, S. Meldgaard, and J. Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *Proceedings of Conference of Theory of Cryptography, Providence, USA*, pages 144–163, 2011.

[11] S. Devadas, M. van Dijk, C.W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. *IACR Cryptology ePrint Archive*, 2015:5, 2015.

[12] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *Proceedings of USENIX Security Symposium, Bellevue, WA, USA*, pages 269–283, 2012.

[13] Christopher W. Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM. *IACR Cryptology ePrint Archive*, 2015: 1065, 2015.

[14] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: round-optimal oblivious RAM with applications to searchable encryption. *IACR Cryptology ePrint Archive*, 2015:1010, 2015.

[15] C. Gentry, K.A. Goldman, S. Halevi, C.S. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Proceedings of Privacy Enhancing Technologies*, pages 1–18, 2013.

[16] 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.

[17] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996. ISSN 0004-5411.

[18] 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.

[19] 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.

[20] 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.

[21] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 513–524, 2012.

[22] Yuval Ishai and Eyal Kushilevitz. Improved upper bounds on information-theoretic private information retrieval (extended abstract). In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, pages 79–88, 1999.

[23] 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.

[24] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient ram-model secure computation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 623–638, 2014.

[25] S. Lu and R. Ostrovsky. Distributed Oblivious RAM for Secure Two-Party Computation. In *TCC*, pages 377–396, 2013.

[26] T. Mayberry, E.-O. Blass, and A.H. Chan. Efficient Private File Retrieval by Combining ORAM and PIR. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, USA, 2014.

[27] T. Moataz, E.-O. Blass, and G. Noubir. Recursive Trees for Practical ORAM. In *Proceedings of Privacy Enhancing Technologies Symposium*, pages 115–134, Philadelphia, USA, 2015.

[28] T. Moataz, T. Mayberry, and E.-O. Blass. Constant Communication ORAM with Small Blocksize. In *Proceedings of Conference on Computer and Communications Security*, pages 862–873, 2015.

[29] T. Moataz, T. Mayberry, E.-O. Blass, and A.H. Chan. Resizable Tree-Based Oblivious RAM. In *Proceedings of Financial Cryptography and Data Security*, pages 147–167, San Juan, Puerto Rico, 2015. ISBN 978-3-662-47853-0.

[30] 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.

[31] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of Eurocrypt, Prague, Czech Republic*, pages 223–238, 1999.

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

[33] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[34] L. Ren, C.W. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path Oblivious-RAM. In *Proceedings of High Performance Extreme Computing Conference*, pages 1–6, Waltham, USA, 2013.

[35] L. Ren, C.W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants Count: Practical Improvements to Oblivious RAM, 2014. IACR Cryptology ePrint Archive 997.

[36] Adi Shamir. How to share a secret. *Communications of the ACM*, 22 (11):612–613, 1979.

[37] 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.

[38] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 247–258, 2013.

[39] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 253–267, 2013.

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

[41] 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.

[42] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 850–861, 2015.

[43] X.S. Wang, Y. Huang, T.-H.H. Chan, A. Shelat, and E. Shi. SCORAM: Oblivious RAM for Secure Computation. In *Proceedings of Conference on Computer and Communications Security*, pages 191–202, Scottsdale, USA, 2014.

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

[45] 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.