Constant Communication ORAM without Encryption

Tarik Moataz[1], Erik-Oliver Blass[2], Travis Mayberry[3]
[1]Colorado State University & Telecom Bretagne, IMT, tarik.moataz@colostate.edu
[2]Airbus Group Innovations, erik-oliver.blass@airbus.com
[3]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 additively homomorphic encryption on both the server and the client. We present two new hybrid ORAM constructions that combine ORAM with Private Information Storage. We store and access individual ORAM buckets with PIS. As a result, our first ORAM features $\widetilde{O}(\log N)$ communication complexity and a small block size of $\Omega(\log^3 N)$ bit. The second ORAM features optimal $O(1)$ communication complexity and $\Omega(\log^4 N)$ bit block size. Both ORAMs have constant client-side memory complexity. The highlight of our approach is that neither client nor server are required to perform any encryption. The above properties make our ORAMs extremely lightweight, suitable for deployment even on resource-constrained devices. In addition to a theoretical analysis, we also implement our ORAMs to show their practicality and compare to related work.**

I. Introduction

Oblivious RAM is a very powerful tool that has been the subject of substantial research efforts over the last few years. It allows a client to outsource data to an untrusted server while hiding the pattern of accesses that the client performs on this data. The classic measure of the efficiency of an ORAM scheme is its communication overhead, how much extra data must be transferred between the client and the server per access in order to gain access pattern privacy.

Recently, research has achieved optimal $O(1)$ communication complexity in the number of blocks stored in the ORAM [7, 21]. Unfortunately, these schemes achieve good communication complexity by leveraging server computation which may be prohibitively expensive. In current schemes, homomorphic ciphers like Paillier [24] are used on the server to perform oblivious shuffling. This causes server computation exceeding several minutes for every client access which (in terms of time) outweighs any savings in communication complexity.

Our contribution: We show that homomorphic encryption used in related work can be replaced by much faster operations if the single server assumption is relaxed to allow for a small number (as few as 4) of non-colluding servers. This allows for ORAM with optimal $O(1)$ communication complexity *and* efficient server computations in the of order of milliseconds instead of minutes. This makes ORAM possible for low-powered clients that would not even be required to compute encryptions. Additionally, we answer a longstanding open question posed in [23] by showing that an ORAM scheme can be made information-theoretically secure. Our construction relies only on a non-collusion assumption, but no computational assumption.

We combine recent advances in constant communication ORAM with Private Information Storage (PIS) and present two new ORAM constructions: NE-ORAM and CNE-ORAM. Instead of naïvely storing the whole ORAM tree in PIS for information-theoretic security, our idea is to store and access buckets separately. This results in low (constant) communication complexity, memory complexity, and small block size. Table I presents technical highlights of NE-ORAM and CNE-ORAM compared to related work.

A. Motivating example

Considering the block size and the number of homomorphic operations, C-ORAM [21] represents an efficient constant communication ORAM in the semi-honest model. That is in order to access one block obliviously, only one block is retrieved from the server. This is made possible by allowing the server to leverage some computations on the ORAM structure. 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 all possible gains in communication. For a clear view on this issue, we take results from C-ORAM's evaluation [21]. Using Pailler as their additive homomorphic encryption, for $N = 2^{20}$, the computation in C-ORAM takes around 10 minutes. This does not even take into account the time to produce the CPIR queries which are based on homomorphic encryption of a logarithmic number of vectors. We expect that the overall delay can exceed by far the 10 minutes needed for pure computation on the server side. Lattice based schemes such as NTRU might decrease computation on the server side to be of the order of 5 to 10 seconds. While this appears to be an interesting approach, it is still unclear how to fix NTRU parameters to have both efficiency and well-defined security.

B. Construction Overview

We present a high level overview of our two new ORAM constructions NE-ORAM and CNE-ORAM. Both improve the state of the art on ORAMs with constant client memory which makes them especially appealing for resource-constrained devices. Our first construction, NE-ORAM has communication

complexity that is logarithmic in the number of elements N and requires small blocks of size $\log^3 N$. Our second and main ORAM construction CNE-ORAM extends NE-ORAM and achieves constant communication complexity. However, NE-ORAM requires slightly larger blocks of size $\log^4 N$. We describe both ORAMs, as they could target different application settings, depending on real-world block size and efficiency requirements.

1) NE-ORAM overview: Let $\widetilde{O}$ be asymptotic worst-case behavior hiding a $\mathsf{poly}(\log\log N)$ factor. NE-ORAM (*No Encryption ORAM*) is an information-theoretic ORAM combining recent work in tree-based ORAMs with Private information storage (PIS). NE-ORAM is a constant client memory ORAM with communication complexity $\widetilde{O}(\log N)$ for a block size in $\Omega(\frac{\log^3 N}{(\log\log N)^3})$. NE-ORAM builds on the tree-based ORAM by Shi et al. [28] and a PIS construction by Ostrovsky and Shoup [23].

To remove encryption and achieve information-theoretic security, we employ multiple non-colluding servers to store the ORAM. To make that efficient, the main idea behind NE-ORAM is to exploit inherently leaked access information of tree-based ORAMs. Reading an element in an ORAM tree requires downloading a logarithmic number of buckets that reside on a specific path. This path is known to the server. Similarly, performing an eviction will access specific buckets on every level of a path, and this pattern is also known to the server.

Therefore, instead of the naïve approach of replicating the entire ORAM tree on multiple servers, our idea is to replicate individual buckets. We then use PIS to obliviously access each bucket. This allows NE-ORAM to improve communication complexity. For block size B and a bucket size of (at least) $\log N$, accessing a single bucket in Shi et al. [28]'s ORAM would be in $O(\log N \cdot B)$. Using PIS, accessing a single bucket will only cost us $\widetilde{O}(B)$.

2) CNE-ORAM overview: CNE-ORAM (*Constant-Communication-Complexity No Encryption ORAM*) integrates recent work on constant communication complexity ORAM [7, 21] to propose a new constant communication ORAM without encryption. NE-ORAM offers $O(1)$ bandwidth overhead for a blocks size in $\Omega(\log^4 N)$ and constant client memory. Moreover, server computations are simple XOR operations which are very cheap compared to expensive homomorphic operations of related work [7, 21]. CNE-ORAM builds on top of NE-ORAM in the sense that it also uses replication on multiple servers for information-theoretic security. For CNE-ORAM, we adopt an oblivious merge technique [21] to our setting with private information storage.

II. Background

We briefly present an overview about the main primitives that we are going to base our work on. That is, the ORAM by Shi et al. [28], IT-PIR by Chor et al. [4], PIS by Ostrovsky and Shoup [23], and the oblivious merge technique by Moataz et al. [21].

A. Tree-based ORAM

An ORAM allows two operations on an outsourced memory. $\mathsf{Read}(\mathsf{a})$ reads from and $\mathsf{Write}(\mathsf{a},\mathsf{data})$ writes to a block of data given the memory address a and the new value to be written data. 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 $\lambda \in \Omega(\log N)$ blocks, giving a failure (overflow) probability of $2^{-\lambda}$ during later eviction. Each leaf is associated to a leaf tag $\mathsf{tag} \in \{0,1\}^{\log N}$. 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 $\log N$ 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 formulated as a $\mathsf{ReadAndRemove}$ operation, which additionally removed the block from the tree, followed by an Add operation such as in [28].

- $\mathsf{ReadAndRemove}(\mathsf{a})$: Given address a, the client fetches leaf tag tag from the position map. Given tag, the client downloads path $\mathcal{P}(\mathsf{tag})$ that starts from the root and ends with leaf tag. The client decrypts the path $\mathcal{P}(\mathsf{tag})$, and retrieve the block searched for. The block is replaced by a dummy block. Finally, the client re-encrypts all buckets and uploads them to 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.
- $\mathsf{Add}(\mathsf{a},\mathsf{data})$: 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 then client adds the block to the root.

To prevent the root bucket from overflowing, an eviction process is necessary to percolate real blocks towards their tagged leaves. The eviction process, as defined in [28], accesses ν buckets per level. For each bucket, the client reads a block and writes it to the bucket's child that is closer to the block's leaf tag t. To preserve obliviousness, the client needs to write a dummy block in child's sibling as well.

Complexity Analysis: Blocks in the position map must have size $\log N$ bit. To fetch a tag from the position map, a total of $O(\log^4 N)$ bit are fetched: $\log N$ recursive trees with height $\log N$ and buckets with $\log N$ blocks of size $\log N$. To then read the data block, $O(\log^2 N \cdot B)$ bit are also transferred to the client. Thus in total, $O(\log^2 N \cdot B + \log^4 N)$ bit, i.e., $O(\log^3 N)$ blocks are communicated between the server and the client.

B. Information-Theoretic Private Information Retrieval

Information-theoretic Private information retrieval (IT-PIR) is a cryptographic primitive introduced by Chor et al. [4]. In

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

TABLE I: Comparison of recent ORAMs, block size B in function of the number of blocks N, client memory in blocks, communication complexity in blocks, $\widetilde{O}$ hides $\mathsf{poly}(\log\log N)$ factors

Scheme	Block Size B	Client Memory	Communication	# homomorphic scalar multiplications	Security	# Servers
Path ORAM [30]	$\Omega(\log^2 N)$	$O(\log N)$	$O(\log N)$	–	Computational	1
Onion ORAM [7]	$\Omega(\log^6 N)$	$O(1)$	$O(1)$	$\Theta(B\lambda \log N)$	Computational	1
C-ORAM [21]	$\Omega(\log^4 N)$	$O(1)$	$O(1)$	$\Theta(B\lambda)$	Computational	1
NE-ORAM	$\Omega(\frac{\log^3 N}{(\log\log N)^3})$	$O(1)$	$\widetilde{O}(\log N)$	–	Information-Theoretic	4
CNE-ORAM	$\Omega(\log^4 N)$	$O(1)$	$O(1)$	–	Information-Theoretic	4

Input: Position $\mathsf{pos} \in \{1,\dots,N\}$
Output: Block data stored at pos
// Client, generate PIR vectors
Set $\mathsf{adr}[\mathsf{pos}] = 1$ and $\forall i \neq \mathsf{pos} : \mathsf{adr}[i] = 1$;
$\mathsf{vect}_1 \xleftarrow{\$} \{0,1\}^N$;
$\mathsf{vect}_2 := \mathsf{Vect}_1 \oplus \mathsf{adr}$;
Send vect_1 to Server 1 and vect_2 to Server 2;
$\mathsf{rsl}_1 = \mathsf{rsl}_2 = 0^N$;
for i **from** 0 **to** N **do**
  // Server 1
  if $\mathsf{vect}_1[i] = 1$ **then** set $\mathsf{rsl}_1 = \mathsf{rsl}_1 \oplus B_i$;
  // Server 2
  if $\mathsf{vect}_2[i] = 1$ **then** set $\mathsf{rsl}_2 = \mathsf{rsl}_2 \oplus B_i$;
end
Send rsl_1 and rsl_2 to client;
// Client
$\mathsf{data} := \mathsf{rsl}_1 \oplus \mathsf{rsl}_2$;

Algorithm 1: Linear IT-PIR for two servers

contrast to ORAM, IT-PIR does not require the outsourced database to be encrypted to hide access pattern. However, only read operation are possible in IT-PIR. To be able to have 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 [1, 2, 17] and many derivatives. For completeness sake, we also mention that there exists work preserving information theoretic security against up to $t < k$ colluding servers [8].

In this paper, we are particularly interested in retrieving (large) blocks of data, not single bit. For this purpose, linear constructions achieve constant communication overhead, yet for different lower bounds on the block size. It turns out that the basic construction by Chor et al. [4] is sufficient for the needs of our second construction CNE-ORAM.

This construction is shown in Algorithm 1. 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-theoretic secure PIR is linear in N.

For our first ORAM construction NE-ORAM, however, we need a slightly more involving IT-PIR mechanism. Due to space limitations, we do not present details here, but only repeat the main results from [4]. The first result states the communication complexity to obliviously read a single bit, the second shows how we can save communication complexity when reading a block of B bit.

Theorem 1: (Corollary 2 of [4]) For k servers each holding N bit of data, there exists a construction to retrieve a single bit with communication complexity $O(k \cdot \log k \cdot N^{\frac{1}{\log k}})$.

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.

Theorem 2: (Corollary 12 of [4]) 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.

Theorem 3: (Corollaries 13 and 14 of [4]) For any constant $k \geq 2$ and for any $B \geq \log k \cdot N^{\frac{1}{\log k}}$, there exists an IT-PIR$(B, N, k)$ construction with communication complexity $O(B)$ bit.

Remark: Many improved constructions exist for IT-PIR that lower the block size bound. For example, $B \geq N^{\frac{1}{k}}$ instead of $B \geq \log k \cdot N^{\frac{1}{\log k}}$, see [4]. However for NE-ORAM and CNE-ORAM, the above two IT-PIR constructions for block retrieval are sufficient. Note that we do not use the IT-PIR scheme from Theorem 3 for our second ORAM, CNE-ORAM, but the simple one from Algorithm 1. While this IT-PIR offers better communication complexity, it cannot be applied to our oblivious permutation – this will become clear later in Section IV.

C. Private Information Storage

IT-PIR only supports reading. To support writing, Ostrovsky and Shoup [23] introduce Private Information Storage (PIS) transforming any read-only IT-PIR to also support writes. All PIS have $O(1)$ memory complexity on the client side.

Theorem 4: (Theorem 2 of [23]) For any $k \geq 2$, if there is a k-server IT-PIR scheme IT-PIR(B, N, k) on N blocks, then there is a $2 \cdot k$-server private read/write scheme on N blocks with communication complexity $O(\log^3 N)$ times the communication complexity of IT-PIR(B, N, k).

Our final result is therefore Corollary 1.

Corollary 1: For any $k \geq 2$ and $B \geq \log k \cdot N^{\frac{1}{\log k}}$, there is a $2 \cdot k$-server private read/write scheme on N blocks with communication complexity $O(B \cdot \log^3 N)$.

For such a PIS scheme, we denote by $\mathsf{PIS}(\mathsf{pos}, \mathsf{DB}, k, \perp)$ the read operation and by $\mathsf{PIS}(\mathsf{pos}, \mathsf{DB}, k, B)$ the write operation of a block at position pos from k servers on database DB. A PIS protocol applied to a plaintext database DB refers to the transformation applied on the plaintext database. This transformation is a special kind of data encoding, we refer to Ostrovsky and Shoup [23] paper for more details about the construction. In the remaining of the paper, a PIS *database* denotes the resulting encoded database.

D. Oblivious Merge

Given two ORAM buckets, each containing a number of "empty", "real", and "noisy" blocks, an oblivious merge algorithm [21, 22] outputs a permutation that will merge both buckets into one. Noisy blocks were real blocks that are either percolated to the wrong path as results of [21] eviction, or previously the output of a read operation. This permutation changes the transposition of blocks of the first bucket such that no two real blocks will collide while merging. For the adversary, an oblivious merge permutation is computationally indistinguishable from a randomly chosen permutation on buckets. So, 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 [21]. We present details about oblivious merge technique in Algorithm 2. The client first gets the distribution of real, empty and noisy blocks in buckets A and B. The client determines the empty blocks in B, and randomly selects a number of empty positions for the real blocks in A to be mapped to. Analogically, the client does the same for the real blocks in B. The client then determines the noisy blocks in B and randomly selects the spots for the noisy blocks in A to be mapped to. If there are more noisy blocks in A than B, they are then mapped randomly to empty blocks in B. Finally, the lasting blocks are mapped randomly together.

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
   π
```

Let x_1, x_2 be the number of empty and noisy slots in A;
Let y_1, y_2 be the number of full and noisy slots in B;
$d_1 = x_1 - y_1$;
$d_2 = x_2 - y_2$;
for i **from** 1 **to** $\mu \cdot z$ **do**
  case $B[i]$ *is full* $z \xleftarrow{\$}$ all empty slots in A ;
  case $B[i]$ *is noisy*
    if $d_2 > 0$ **then**
      $z \xleftarrow{\$}$ all noisy slots in A;
      $d_2 = d_2 - 1$;
    else
      $z \xleftarrow{\$}$ all empty slots in A;
    end
  end
  case $B[i]$ *is empty*
    if $d_1 > 0$ **then**
      $z \xleftarrow{\$}$ all non-assigned slots in A;
      $d_1 = d_1 - 1$;
    else
      $z \xleftarrow{\$}$ all full slots in A;
    end
  end
  $\pi[i] = z$;
  $A[z] =$ assigned;
end
return π;

Algorithm 2: $\mathsf{GenPerm}(A, B)$, oblivious permutation generation [21]

E. Secret Sharing

For our two protocols, 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 the shares. There are many schemes with different properties and approaches, e.g., [3, 27], 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.

III. NE-ORAM CONSTRUCTION

Similar to Shi et al. [28], NE-ORAM stores N blocks in a tree with N leaves. However, in contrast to regular ORAM that stores all buckets/nodes on one server, we store all buckets on a total of 4 non-colluding servers ($k = 2$) using the PIS of Corollary 1. Therefore, each bucket with its λ blocks is represented as a PIS database.

For now, assume that the position map that maps each address to its tag is available on the client. As mentioned before, the position map will be stored recursively on the server to have constant memory complexity on the client.

One of the main building blocks of our NE-ORAM construction are headers. In addition to λ blocks, each bucket also stores two types of headers. First, header^1 is a λ by $\log N$ bit two-dimensional array. Each row i stores the address of block i in the bucket. Second, header^2 is another λ by $\log N$ two-dimensional array where row i represents the leaf tag of block i. Both headers are accessed using PIS reads/writes on k servers. That is, both headers are stored as a PIS database over the 4 servers.

Given the address of a block at address adr, the client first fetches leaf tag tag from the position map. The leaf tag defines a path of nodes $\mathcal{P}(\mathsf{tag}) = \{\mathcal{P}(0, \mathsf{tag}), \ldots, \mathcal{P}(\log N, \mathsf{tag})\}$. Starting from the root, the client uses PIS to retrieve header_i^1 from each bucket $\mathcal{P}(i, \mathsf{tag})$. With this header, the clients checks whether the block at address adr exists in this bucket. If adr exists in row j in header_i^1, the client uses PIS to read block j from this bucket and writes a dummy block back into the bucket as a replacement. Otherwise, if the address does not exist, the client samples a random position $j \xleftarrow{\$} [\lambda]$, uses PIS to retrieve block j, and writes it back. This procedure is iteratively performed on all buckets $\mathcal{P}(i, \mathsf{tag})$ on the path to tag. Once the block is retrieved, the client also adds it to the root with a PIS write. Writing to a block is similar to reading:

the old block is read and removed from the tree, and the updated block is added to the root. We present details of our read/write "access" operation in NE-ORAM in Algorithm 3. Parameter op can be either a read or a write.

After every read or write, the client has to evict blocks towards leaves. NE-ORAM evicts along the lines of Shi et al. [28]. The difference is that, instead of downloading/uploading an entire bucket, only the headers are downloaded, and PIS read and write are performed on the bucket. At a higher level, eviction works as follows: for each level, ν buckets are randomly selected. For each bucket, a real element is evicted towards the corresponding child. The selection of real elements are based on the header information. In order to preserve obliviousness, both children of any selected node among the ν evicted has to be accessed. Based on the leaf tag tag, either a dummy or a real block is written. This process is repeated for all tree levels. We show details of NE-ORAM's eviction in Algorithm 4.

Remark: One might notice that applying PIS to access one header using Ostrovsky and Shoup [23]'s construction is more expensive than the naïve construction (discussed later). From an asymptotic point of view, using the naïve construction will improve over our scheme using PIS. Our choice is simply to preserve clarity and the same PIS construction for both buckets and headers. In practice, we argue that using naive PIS for headers would be cheaper and easier to implement.

Input: Operation op, address adr, block block
Output: Block data associated to address addr
// Fetch tag value from position map
1 $\mathsf{tag} = \mathsf{posMap}(\mathsf{adr})$;
2 $\mathsf{posMap}(\mathsf{adr}) \xleftarrow{\$} [N]$;
3 **for** i **from** 0 **to** $\log N$ **do**
  // Download header^1
4 $\mathsf{header}_i^1 = \mathsf{PIS}(1, \mathsf{header}_\mathsf{i}^1, \mathsf{k}, \perp)$;
5 **for** j **from** 1 **to** λ **do**
    // Search if adr exists in header^1
6 **if** $\mathsf{header}_\mathsf{i}^1[\mathsf{j}] = \mathsf{adr}$ **then**
7 set $\mathsf{pos} = j$;
      // Read the block searched for
8 $\mathsf{data} = \mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{tag}, \mathsf{i}), k, \perp)$;
      // Replace the real block with a dummy and update the header
9 $\mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{tag}, \mathsf{i}), k, \mathsf{dummy})$;
10 update $\mathsf{headers}_\mathsf{i}^1[\mathsf{j}] = 0$;
11 **end**
12 **end**
    // If the block was not found in the bucket
13 **if** $\mathsf{pos} = 0$ **then**
14 set $\mathsf{pos} \xleftarrow{\$} [\lambda]$;
15 $\mathsf{data}^* = \mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{tag}, \mathsf{i}), k, \perp)$;
16 $\mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{tag}, \mathsf{i}), k, \mathsf{data}^*)$;
17 **end**
18 $\mathsf{PIS}(1, \mathsf{header}_\mathsf{i}^1, \mathsf{k}, \mathsf{header}_\mathsf{i}^1)$;
19 **end**
20 **if** $\mathsf{op} = \mathsf{write}$ **then** set $\mathsf{data} = \mathsf{block}$;
  // Add the real block to the root bucket and update the header
21 $\mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{tag}, 0), k, \mathsf{data})$;
22 $\mathsf{PIS}(1, \mathsf{header}_\mathsf{i}^1, \mathsf{k}, \mathsf{header}_\mathsf{i}^1)$;
23 $\mathsf{Evict}(\nu)$;

Algorithm 3: Access(op, adr, block): NE-ORAM access operation

A. NE-ORAM Complexity Analysis

In this section, we analyze NE-ORAM's communication complexity. Our analysis is divided into two parts. First, we analyze access operations (Algorithm 3), and second, we analyze eviction (Algorithm 4).

Headers header^1 and header^2 are both two-dimensional arrays of size $\lambda \cdot \log N$ bit. To access a header, we retrieve it as a whole upload it as a whole. Communication complexity for the headers is in $O(\lambda \cdot \log N)$ bit. Based on previous overflow analysis [28], $\lambda \in O(\log N)$, so headers require $O(\log^2 N)$ communication complexity.

The bucket contains $\lambda \in O(\log N)$ blocks, each of size B bit. With $k = 2, B \geq \log N$, Corollary 1 yields a communication complexity for a bucket of $O(B \cdot (\log \log N)^3)$.

Since an access implies reading/writing a logarithmic number of buckets with headers, the access communication complexity is in $O(\log N \cdot (\log^2 N + B \cdot (\log \log N)^3))$.

For the eviction, we need to access ν buckets per level. For each bucket, the children need to be accessed, too. So, $3 \cdot \nu$ buckets in total are accessed for every level. Based on previous analysis [22], ν is a constant and has to be equal to 4. In conclusion, eviction has the same communication complexity as block access.

The position map is recursively stored on the server, as a sequence of trees of increasing (doubling) size. The number of such trees is logarithmic in N. Asymptotically, summing over the linearly increasing height is equivalent to assuming all trees to have height $\log N$. In each tree, blocks have size equal $\log N$ bit (in practice, block sizes vary from one tree to another). To retrieve a leaf tag, the client needs communication in $O(\log N \cdot (\log^3 N + \log^2 N \cdot (\log \log N)^3))$.

In conclusion, communication complexity of NE-ORAM is

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

If $B \in \Omega(\frac{\log^3 N}{(\log \log N)^3})$, the above quantity is equivalent to

$$\widetilde{O}(B \cdot \log N).$$

So far, we have implicitly considered the number of servers required for PIS to be equal 4 without giving any further explanations. Moreover, we have considered the number of servers constant and therefore not considered it in our complexity analysis above. As of Theorem 3, block size B must be larger than $\log k \cdot (B \cdot \lambda)^{\frac{1}{\log k}}$ to have an $O(B)$ IT-PIR and Corollary 1 to be valid. That is, in order for our bandwidth analysis to be valid, the block size $B \geq \log k \cdot (\lambda)^{\frac{1}{\log k}}$. Fixing $k = 2$, the block size $B \geq \lambda$. On the other hand, we have shown that in NE-ORAM, we require the block size to be

Input: Eviction rate ν
for i **from** 0 **to** $\log N$ **do**
  for j **from** 1 **to** ν **do**
    set $\mathsf{ind} \xleftarrow{\$} [2^i]$;
    // Download headers of bucket $\mathcal{P}(\mathsf{ind}, i)$
    $\mathsf{header}_\mathsf{i}^1 = \mathsf{PIS}(1, \mathsf{header}_\mathsf{i}^1, \mathsf{k}, \perp)$;
    $\mathsf{header}_\mathsf{i}^2 = \mathsf{PIS}(1, \mathsf{header}_\mathsf{i}^2, \mathsf{k}, \perp)$;
    set $\mathsf{pos} \xleftarrow{\$}$ Real block positions;
    // Update headers
    update $\mathsf{header}_\mathsf{i}^1[\mathsf{pos}] = 0$ and $\mathsf{header}_\mathsf{i}^2[\mathsf{pos}] = 0$;
    // Retrieve the block to evict
    $\mathsf{data} = \mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{ind}, \mathsf{i}), k, \perp)$;
    // Replace the evicted block with a dummy
    $\mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{ind}, \mathsf{i}), k, \mathsf{dummy})$;
    // Determine the tag of the leaf to be evicted to
    $\mathsf{tag} = \mathsf{headers}^2{}_{\mathsf{pos}}$;
    download headers of bucket $\mathcal{P}(\mathsf{ind}, i+1)$ and $\mathcal{P}(\mathsf{ind}+1, i+1)$;
    set $\mathsf{pos} \xleftarrow{\$}$ Dummy block positions;
    // Write the real/dummy block in the children buckets
    if $\mathsf{tag}_{\mathsf{i}+1} = 0$ **then**
      $\mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{ind}, \mathsf{i}+1), k, \mathsf{dummy})$;
      $\mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{ind}+1, \mathsf{i}+1), k, \mathsf{data})$;
    else
      $\mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{ind}+1, \mathsf{i}+1), k, \mathsf{dummy})$;
      $\mathsf{PIS}(\mathsf{pos}, \mathcal{P}(\mathsf{ind}, \mathsf{i}+1), k, \mathsf{data})$;
    end
    update headers accordingly;
  end
end

Algorithm 4: Evict(ν): NE-ORAM evict operation

$B \in \Omega(\log^2 N)$ which is clearly larger than the lower bound of blocks $\lambda \in O(\log N)$. As PIS requires $2 \cdot k$ server for an IT-PIR with k servers, the total number of servers is 4.

IV. CNE-ORAM

In CNE-ORAM, we augment C-ORAM [21] with IT-PIR. CNE-ORAM inherits C-ORAM's major properties, such as its eviction technique, bucket size, tree structure, and more importantly the oblivious merge technique. CNE-ORAM main idea is to integrates IT-PIR's read techniques into CNE-ORAM. This is challenging, as the oblivious merge must be adopted to function in our no-encryption framework. In C-ORAM, the oblivious merge makes crucial use of additively homomorphic encryption, which we have to find a suitable information-theoretic equivalent for.

For CNE-ORAM, we use the basic IT-PIR construction of Algorithm 1 and the secret sharing technique of Section II-E. While this basic PIS construction has communication complexity linear in the number of blocks, we show how to efficiently use it in a tree-based ORAM setting with oblivious merge. This will lead to constant communication complexity per access in total.

Overview: As a start, 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. This approach trivially gives information theoretic security, i.e., without collusion, a server cannot learn anything from their share. As we will be using IT-PIR on each share separately, we need to introduce two more 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.

We now present full details about how to create and access shares Tree_1 and Tree_2.

Details: CNE-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. The first two headers are λ by L two dimensional arrays. Similarly to NE-ORAM, 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, cf. Section II-D.

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 [21]. We will give more details about the choice of these parameters in the analysis, cf. Section V. 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 $\mathsf{data}_1 \xleftarrow{\$} \{0,1\}^{\log B}$. We store data_1 on two servers s_1, s_1', respectively, and data_2 on servers s_2, s_2', respectively. For clarity sake, 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, let us again assume that the position map is stored on the client side. As before, we recursively outsource the position map in smaller CNE-ORAM structures [28] and add a logarithmic factor in our complexity analysis later.

To read a block at address adr, the client first fetches leaf tag tag from the position map. Second, the client instructs servers to download all headers of buckets on the path $\mathcal{P}(\mathsf{tag})$. For each header, the client receives back 4 bit strings b_1, b_2, b_3, b_4 and can reassemble the header by computing $b_1 \oplus b_2 \oplus b_3 \oplus b_4$. Using the headers, 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 $\mathsf{vect}_1 \xleftarrow{\$} \{0,1\}^{\lambda \cdot L}$. 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 bit strings received. Finally, the client has to upload back the block to the root, and sets the previous position of the retrieved

block to noisy. For this, the client has to download all headers, generates new shares while updating both the header of the root and the header of the block that has been accessed. Note that generating new shares hides from the server which header has been updated. As in [21], one block from the accessed leaf has to be refreshed in case it contains a noisy block. This is performed in a deterministic order. For the same leaf, a block is never accessed twice before accessing all other blocks.

After χ read operations, the client needs to evict real blocks from the root bucket. First, the client downloads the entire root bucket, shuffles the position of real elements and adds a number of empty blocks. The client uploads the root bucket to the server. The client then downloads all headers, applies the oblivious merge algorithm to output L oblivious merge permutations. The server 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.

Atomic insertion: The eviction process as described above is performed after every χ read operations. This will only provide an amortized constant communication scheme. Fortunately, we can de-amortize the eviction by the following trick. We know that after every evict operation, the root bucket is empty. Now, instead of inserting the real blocks after every read and then performing the shuffle operations on the client side, we upload after every read ϕ blocks, where $\phi - 1$ equals the number of empty blocks needed to handle the noisy blocks in lower levels. While the eviction is still performed after every $\chi = O(\lambda)$ operations, its cost in terms of bit is now equal to a read operation. Thus, CNE-ORAM offers a worst-case constant communication overhead.

We give further details about access operation and eviction in Algorithms 5 and 6.

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) $\xleftarrow{\$}$ $[N]$;
3 **for** i **from** 0 **to** L **do**
4 download header_i^1 and header_i^3 ;
5 **for** j **from** 1 **to** λ **do**
// Search if adr exists in 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**
13 $\mathsf{ptr}[i] := \delta_{i,\mathsf{pos}}$;
14 **end**
15 $\mathsf{vect}_1 \xleftarrow{\$} \{0,1\}^{L\cdot\lambda}$;
16 $\mathsf{vect}_2 := \mathsf{Vect}_1 \oplus \mathsf{ptr}$;
// Computation on servers side
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
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 ;
// Atomic 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** $\mathsf{ctr} = 0 \ mod(\chi)$ **then** Evict(st) ;
26 set ctr := ctr + 1;

Algorithm 5: Access(op, adr, block, ctr, st): CNE-ORAM access operation

A. CNE-ORAM correctness

We need to show that the read operation, cf. Algorithm 5, will output the "correct" block data. Also, we need to show that merging buckets in Algorithm 6 preserves the values of real blocks.

In Algorithm 5 lines 17-20, the server performs XOR operations over blocks for which the IT-PIR vector contains a 1. We know that CNE-ORAM's tree is stored over four servers. Two servers storing two shares, while the other two store exact duplicates of each share. In order to retrieve a block, the client needs to retrieve the block from all four servers. Formally, 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 the duplicates r_i and $\mathsf{data}_i \oplus r_i$. Now, each read operation, the path $\mathcal{P}(\mathsf{pos})$ has to be accessed in order to retrieve the desired block, see Algorithm 5 lines 1-2. Now, given this path, we need to identify the bucket that contains the block we are looking for. Thus, all headers needs to be downloaded, lines 3-11, in order to determine the block position. Given this position, the client generates four IT-PIR vectors 15-16 and send each to the corresponding server. For two servers among the four, the block position, pos, in the IT-PIR vector is set to one, because we need to retrieve both shares. For the the other two vectors, 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 $\{\mathcal{P}_1(\mathsf{tag}), \cdots, \mathcal{P}_{\lambda\cdot L}(\mathsf{tag})\}$. The client retrieves from each server $\mathsf{rsl}_{i,j}$ for $i, j \in [2]$ such that

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

Input: Eviction state st
for i **from** 0 **to** $L-1$ **do**
  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
  set $\pi \leftarrow \mathsf{GenPerm}(\mathsf{H_i}, \mathsf{H_{i+1}})$;
  // Merge the parent and destination bucket
  $\mathcal{P}(\mathsf{st}, i+1) := \pi(\mathcal{P}(\mathsf{st}, i)) \oplus \mathcal{P}(\mathsf{st}, i+1)$;
  if $i < L-1$ **then**
    // Copy the parent bucket into its sibling
    $\mathcal{P}_s(\mathsf{st}, i) := \mathcal{P}(\mathsf{st}, i)$;
  else
    // Merge the last bucket with the sibling leaf
    download headers $\mathsf{H_{i+1}^s}$ from the sibling leaf;
    $\pi \leftarrow \mathsf{GenPerm}(\mathsf{H_i}, \mathsf{H_{i+1}^s})$;
    $\mathcal{P}(\mathsf{st}, i+1) := \pi(\mathcal{P}(\mathsf{st}, i)) \oplus \mathcal{P}(\mathsf{st}, i+1)$;
  end
  update and upload new shares of headers $\mathsf{H_i}$ and $\mathsf{H_{i+1}}$;
  set $\mathcal{P}(\mathsf{st}, i) := \mathbf{0}^{\lambda \cdot B}$;
end

Algorithm 6: Evict(st): CNE-ORAM evict operation

$$\begin{aligned} & \oplus \mathcal{P}_{1,k}(\mathsf{tag}) \oplus \mathcal{P}_{2,k}(\mathsf{tag})\Big) \\ & \oplus \mathcal{P}_{1,\mathsf{pos}}(\mathsf{tag}) \oplus \mathcal{P}_{2,\mathsf{pos}}(\mathsf{tag}) \\ = \;& \mathcal{P}_{1,\mathsf{pos}}(\mathsf{tag}) \oplus \mathcal{P}_{2,\mathsf{pos}}(\mathsf{tag}) \\ = \;& r \oplus r \oplus \mathsf{data} = \mathsf{data} \end{aligned}$$

Now, we need to show that the oblivious merge permutation, when applied to buckets in a specific path, preserves the real blocks. From the C-ORAM [21] correctness proof, we know that there will be always enough space, in the child bucket, for real elements to be percolated in. Here, we are interested in another aspect of correctness. We want to show that xoring up random-looking blocks to each others will not miss up with the value of real and empty block, and a client can always retrieve the correct value later on. Formally, let consider two buckets in two servers containing the two shares $\mathcal{P}_j(\mathsf{tag}, i)$ and $\mathcal{P}_j(\mathsf{tag}, i+1)$, for $j \in [2]$, each containing λ blocks such that:

$$\mathcal{P}_j(\mathsf{tag}, i) = \left(r_{i,1}^j, \cdots, r_{i,\lambda}^j\right)$$

Given the oblivious merge permutation π, we obtain

$$\mathcal{P}_j(\mathsf{tag}, i) \oplus \pi\big(\mathcal{P}_j(\mathsf{tag}, i+1)\big) = \left(r_{i,1}^j \oplus r_{i+1,\pi(1)}^j, \cdots, r_{i,\lambda}^j \oplus r_{i+1,\pi(\lambda)}^j\right)$$

We have 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. Given that a block data exists in the $(i+1)^{\text{th}}$ bucket at the k^{th} position, we need to show that

$$\bigoplus_{j \in [2]} r_{i,k}^j \oplus r_{i+1,\pi(k)}^j = \mathsf{data}$$

Assuming that before merging the buckets, the parent's block was real and the child's block empty. Based on secret sharing, we obtain

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

B. CNE-ORAM bandwidth analysis

In this section, we analyze CNE-ORAM's communication complexity. For constant communication ORAM, we need to determine the lower bound of the block size that CNE-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 $\lambda \in O(\log N)$. These results are going to be further detailed in the security analysis, cf. Section V.

To read a block (Algorithm 5), the client first downloads the headers, sends the IT-PIR query and then downloads the XORed output. Finally, the clients uploads ϕ blocks back to the root bucket. This computes to $4L \cdot \lambda \cdot L + 4L \cdot \lambda + (4+\phi)B = O(\log^3 N + B)$.

To evict a path, the client needs to download the all headers, generate permutations for any two adjacent buckets based on oblivious merge algorithm. The number of required bit for the eviction is $4L \cdot \lambda \cdot L + 4L \cdot \lambda \cdot \log \lambda = O(\log^3 N)$. The eviction is a process that occurs after after $\chi = O(\log N)$ read operations. That is, the amortized number of bit transferred between the client and servers equals $O(\log^2 N)$.

Before any read operation, the client needs to fetch the tag from the position map. The position map is a logarithmic number of CNE-ORAM trees with block size smaller or equal to L. From an asymptotic point of view, it is valid to consider all trees with the same maximal height and block size. That is, the number of trees in the position map, as well as the block size equal L. Every tree in the position map is a CNE-ORAM tree, therefore for a read operation, the number of bit required is similar to the one performed on the main data ORAM except that the block size B is replaced by L. Since L is in $O(\log N)$ and therefore dominated by $O(\log^3 N)$. That is, the read operation is in $O(\log^3 N)$, while the evict equals $O(\log^2 N)$ per tree.

To sum up, the communication overhead equals

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

In order to obtain a constant communication overhead, it would be sufficient to lower bound the block size such that

$$B \in \Omega(\log^4 N).$$

Remark: From an asymptotic complexity point of view, for both NE-ORAM and CNE-ORAM, the position map induces an additional logarithmic factor to the lower bound of the required block size. The position map is responsible for $O(\log^4 N)$ bandwidth overhead, while the access and eviction process only induce a $O(\log^3 N + B)$ communication

complexity overhead. In practice, storing the position map on the client side does not represent a memory bottleneck for fairly large number of elements. For example, for $N = 2^{20}$, the position map will cost only 2 MByte which is small and might even be stored on the client side. Some might argue that storing $N \log N$ bits is contrary to our setting where we target constant memory complexity on the client side. However, this is not always true, since we target a memory complexity constant in the number of *blocks* and not in bits. A block size in CNE-ORAM, $\Omega(\log^4 N)$, can be larger than $N \log N$ for $N \leq 2^{13}$. Storing the position map improves the lower bound on the block size for both NE-ORAM and CNE-ORAM as follows. For NE-ORAM, the minimal block size to have a $\widetilde{O}(\log N)$ communication complexity becomes $\Omega(\frac{\log^2 N}{(\log \log N)^3})$ bit. For CNE-ORAM, the minimal block size to have constant communication complexity becomes $\Omega(\log^3 N)$ bit.

V. Security analysis

A. Security definition

We will show that an unbounded adversary with non-colluding servers can recover a block's content only with a probability $2^{-|B|}$ which is perfectly secure. However, we stress that due to their tree structure, buckets' in both CNE-ORAM and NE-ORAM can overflow. Thus, their security depends on the bucket overflow rate. Fortunately, bucket overflow, colluding servers, and block security are independent events. In the following security definition, we are going to capture these three aspects. We only define security for CNE-ORAM. Security for NE-ORAM follows along the same lines.

Definition 5.1: Let $\overrightarrow{a} = \{(op_1, d_1, a_1), (op_2, d_2, a_2), \ldots, (op_M, d_M, a_M)\}$ be a sequence of M 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 $op_i = \textit{Add}$, or $d_i = \perp$ when $op_i = \textit{ReadAndRemove}$.

We say that CNE-ORAM is information-theoretically secure *iff*

1) Servers are not colluding
2) Given a security parameter λ, and Overf be the event that a bucket overflows during the access, the probability of overflow $\Pr(\mathsf{Overf}) \leq \mathsf{negl}(\lambda)$.
3) for any computationally unbounded adversary $\mathcal{D}$ and any two same-length sequences $\overrightarrow{a}$ and $\overrightarrow{b}$, access patterns $A(\overrightarrow{a})$ and $A(\overrightarrow{b})$,

$$Pr[\mathcal{D}(A(\overrightarrow{a})) = 1] = Pr[\mathcal{D}(A(\overrightarrow{b})) = 1].$$

First, we will make sure that buckets in CNE-ORAM will not overflow. This implies a parametrization of bucket size λ, height of the tree L, expansion factor ϕ, and eviction rate χ. The eviction rate χ refers to the number of allowed operations before that an eviction occurs. The expansion factor ϕ is a security parameter that increases the bucket size to avoid collisions when using oblivious merge technique. At a higher level, ϕ is a multiplicative stretch factor that increase the number of empty block to be equal to $(\phi - 1)\chi$, and that the overall bucket size $\lambda = \phi \cdot \chi$. Given our atomic eviction technique, we insert in the root one real block after every access operation. Thus the number of real blocks in the root is at most χ before any eviction. Given that our eviction is similar to previous works, we can inherit their analysis. Second, we show that an oblivious merge permutation does not impact ORAM obliviousness. More importantly, we need to show that our new eviction trick, atomic insertion, does not impact CNE-ORAM obliviousness.

Given that oblivious merge with the atomic insertion preserves obliviousness, it is straightforward to show that CNE-ORAM is information-theoretically secure as of Definition 5.1.

B. Overflow probability

CNE-ORAM's eviction is the same as in various previous works [7, 21, 26] and works as follows: a path is first selected following a reverse lexicographic order. After every read operation, real elements are put back in the root. After χ read operations an eviction occurs. In CNE-ORAM, the percolation of real elements in the tree is the same as with previous schemes. That is, for a similar overflow probability, the bucket size are equal to those in previous works [7, 21, 26]. Moreover, the eviction rate and the tree height are similar. Thus, a similar parameterization holds. We only repeat the main results and refer the reader to Devadas et al. [7] for details and proofs.

Theorem 5.1: For eviction rate χ and tree height L, with $\lambda \geq \chi$ and $N \leq \chi \cdot 2^{L-1}$, the probability that a bucket overflows is upper bounded by $e^{\frac{-(2\lambda-\chi)^2}{6\cdot\chi}}$.

To avoid bucket overflows, the probability of overflow has to be small. For an overflow probability negligible in the security parameter z, it is sufficient to choose $\lambda \in \Theta(z)$, $L \in \Theta(\log N)$, $\chi \in \Theta(z)$. In practice, we can choose $z \in \omega(\log N)$.

Besides, in CNE-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 [21] for proof details.

Theorem 5.2: If $\phi \in \Theta(1)$, a real block gets overwritten with a probability in $O(\lambda^{-\lambda})$.

If bucket size $\lambda \in \omega(\log N)$, $L \in \Theta(\log N)$, and $\phi \in \Theta(1)$, the probability that a real block gets overwritten is in $O(N^{-\log\log N})$. Experiments [21] show that empirically $\phi \approx 2.2$ is sufficient.

ϕ is an expansion factor that needs to be adapted to our atomic 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 atomic eviction, we evict less often with more real blocks in the root bucket. Based on results of [21], the probability that a real block gets overwritten at the i^{th} level and j^{th} eviction, $\Pr[R_{i,j}]$, equals

$$\Pr[R_{i,j}] \leq M e^{i+\ln(i\cdot\chi)+4\phi\cdot\chi-\ln(\phi\cdot\chi)\cdot\phi\cdot\chi},$$

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

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

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

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

C. Oblivious merge

It is important to show that the oblivious merge algorithm outputs permutations that are indistinguishable from random permutations for computationally unbounded adversaries. Thus, given a random 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. In C-ORAM [21], authors have shown that with a random root bucket, any adversary has a negligible advantage in distinguishing the output of a random permutation from the output of the oblivious merge algorithm. The negligible, non-zero advantage in the case of C-ORAM is a result of using additive homomorphic and symmetric key encryption. In our case with CNE-ORAM, the adversary has a zero advantage.

We now extend the previous security argument [21]. That is, both buckets are not required to be random. Surprisingly, a slightly weaker assumption is sufficient.

Let GenPerm be the (probabilistic) oblivious merge algorithm. C-ORAM [21] adversarial model is the following: an adversary can chose any pair of two adjacent buckets' headers, and can only get one permutation output by GenPerm for it. The header has to be random from an adversarial view. This operation can be performed an unbounded number of times under the only condition that buckets headers have to be updated or different from any previous adversarial request.

Our new setting is different from the one above, we don't assume that both headers are entirely random for the adversary, and work with the following setting assumptions:

1) the adversary can choose the load of A but not the one of B
2) for both headers, any position can be either empty, real or noisy with a probability > 0 each

CNE-ORAM's atomic eviction satisfies both of these assumptions. First, notice that for the root bucket, we insert after every read operation ϕ blocks such that $\phi - 1$ are empty and one is real in a *random* order. Thus, the adversary knows the load of the root bucket. Second, from an adversarial view, any position can be either empty or real with a probability of $\frac{1}{\phi}$ and $1 - \frac{1}{\phi}$, respectively. These assumptions are weaker than both headers being random. The first assumption represent a worst case. In fact, for all buckets different from the root, an adversary does not have any advantage of finding the exact bucket's load larger than a random guess. That is, the first assumption can be relaxed for buckets different from the root to be: the adversary can choose neither the load of A, nor the one of B.

Let $\mathsf{Load}\big((i,n_A),(j,n_B)\big)$ the event that the number of real and noisy blocks in A and B are i, j, and n_A and n_B. k represents the size of the buckets.

Theorem 5: Let A and B be two buckets such that for all permutations π' from $\{1,\dots,k\}^k \to \{1,\dots,k\}^k$

$$\Pr[\pi \leftarrow \mathsf{GenPerm}(\mathsf{header}(A),\mathsf{header}(B)) = \pi'] = \frac{1}{k!}$$

Proof: We prove the theorem for any two adjacent buckets A and B. To have the case for 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 is the possible 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 GemPerm. 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 combination are called valid combinations of B, and their set is denoted by VC. Moreover, let C be the random variable that captures the combination that bucket B is in before merging. Given that bucket B has j real elements and n_B noisy blocks, the overall number of possible combinations is $\binom{k}{j}\cdot\binom{k-j}{n_B}$. That is, we have for any fixed possible combinations δ_B of bucket B

$$\Pr[C = \delta_B] = \frac{1}{\binom{k}{j}\cdot\binom{k-j}{n_B}}.$$

In conclusion, any block in B can be empty, real or noisy, so all possible combinations on B are possible. Above, for clarity, we made implicitly a simplification that a noisy block in B can be considered as empty, this is why we get all possible combinations over A. Note that this simplification does not affect the results of this proof.

Besides, the probability that random variable Π is equal to a specific permutation π computes to

$$\begin{aligned}\Pr[\Pi=\pi'] &= \sum_{i,j\in[k]} \Pr[\Pi=\pi' \ \wedge\ \mathsf{Load}\big((i,n_A),(j,n_B)\big)]\\ &= \sum_{i,j\in[k]} \Pr[\Pi=\pi' \ \ \mathsf{Load}\big((i,n_A),(j,n_B)\big)]\cdot\\ &\quad \Pr[\mathsf{Load}\big((i,n_A),(j,n_B)\big)]\\ &= \sum_{i,j\in[k]}\sum_{\delta_B\in\mathsf{VC}} \Pr[\mathsf{Load}\big((i,n_A),(j,n_B)\big)]\cdot\\ &\quad \Pr[\Pi=\pi'|\mathsf{Load}\big((i,n_A),(j,n_B)\big),\ C=\delta_B]\cdot\\ &\quad \Pr[C=\delta_B]\end{aligned}$$

Now, we need to identify the probability of the event $E = \{\Pi=\pi'|\mathsf{Load}\big((i,n_A),(j,n_B)\big),\ C=\delta_B\}$. First, note that the event captures the following: 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 an apriori fixed combination of B. This quantity equals $\binom{k-j-n_B}{i}\cdot\binom{k-j-i}{n_A}$. The first term of this

quantity computes the number of combinations to insert the i real blocks in A into blocks in B, that is, $k-j-n_B$. Given these possible combinations, we still need to insert the noise of A into the blocks of B. The blocks in B that can handle the noise are either empty or noisy blocks, thus, we have $k-j-i$ available blocks to handle the n_A noisy blocks of A. Given the number of combinations $\binom{k-j-n_B}{i} \cdot \binom{k-j-i}{n_A}$, we can easily find the valid number of permutations that can be applied over A such that: $n_A! \cdot i! \cdot (k-n_A-i)! \cdot \binom{k-j-n_B}{i} \cdot \binom{k-j-i}{n_A}$.The first three terms of this quantity calculate respectively all permutation over empty, noisy and real blocks. Thus,

$$\Pr[E] = \frac{1}{n_A! \cdot i! \cdot (k-n_A-i)! \cdot \binom{k-j-n_B}{i} \cdot \binom{k-j-i}{n_A}}$$

An important remark at this stage would be that the probability of event E is independent of the valid 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. This is also a counting problem that would be solved as follows: as stated above, the overall possible number of combinations equal $\binom{k}{j} \cdot \binom{k-j}{n_B}$, among those combinations, only few represent those within the set of valid combinations VC. A combination over A is valid iff all real elements in both buckets 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 $\binom{k-i-n_A}{j}\binom{k-i-j}{n_B}$.

That is,

$$|\mathsf{VC}| = \binom{k-i-n_A}{j}\binom{k-i-j}{n_B}.$$

Putting all together, we can verify that

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

■

VI. Evaluation

In order to compare CNE-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 $\omega(\log N)$ 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 note. Figure 1 shows χ versus the number of operations that an instance of CNE-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 many instances of CNE-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 the results in Figure 2. Extrapolating out, we can see that for 50 bits of security, CNE-ORAM requires λ to be approximately 900. This is substantially larger than related constant-communication schemes like C-ORAM [21]. However, the bucket size has very little impact on the cost of CNE-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 3 shows overall query execution time, including network transfer and server computation for CNE-ORAM and the current state of the art scheme Path ORAM [30] 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).

The results show that CNE-ORAM substantially beats Path ORAM, especially as N increases. The cost of CNE-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 CNE-ORAM scales much better for larger N. Note that CNE-ORAM offers constant memory complexity on the client, while Path 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.

Also interesting is the increased efficiency on the client side for CNE-ORAM. The client performs no encryption operations, only XOR on four blocks and generating ϕ blocks of random data using a PRNG. In contrast, Path ORAM clients perform decryption and reencryption on $Z \log N$ blocks. This encryption time is not accounted for in Figure 3 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 CNE-ORAM uniquely attractive to devices with low computational ability.

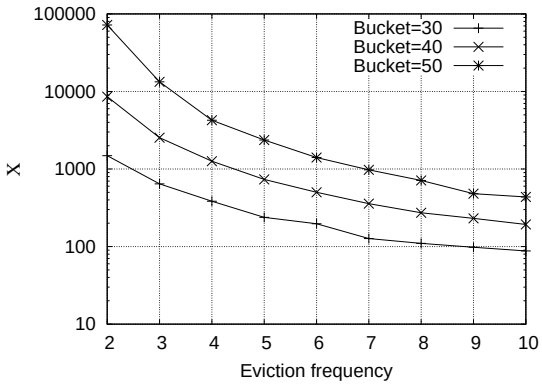

Fig. 1: Eviction rate χ vs number of supported operations

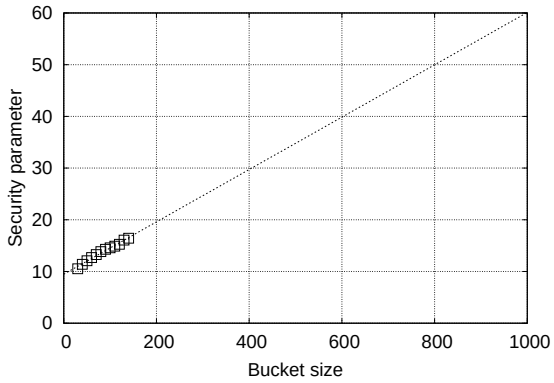

Fig. 2: Size of bucket vs security parameter, with regression line

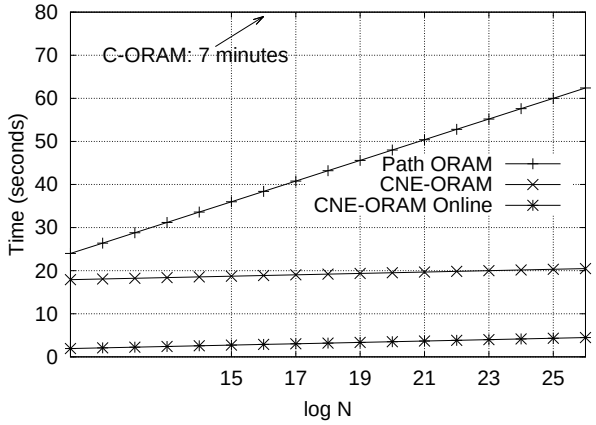

Fig. 3: $\log N$ vs time to perform a query

VII. Related Work

Oblivious RAM goes back to the seminal paper by Goldreich and Ostrovsky [13]. 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 [5, 6, 11–16, 18–23, 25, 26, 28, 30–32]. We briefly review two ORAM categorizations. The first discusses recent advances of schemes with constant client memory complexity, and the second targets schemes with sublinear client memory.

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. Goodrich and Mitzenmacher [14] and Pinkas and Reinman [25] introduced amortize communication complexity in $O(\log^2 N)$, but with linear worst-case communication complexity. Shi et al. [28] 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 [11, 19, 20, 22]. 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. [7] showed how to construct a constant communication ORAM for blocks in $\Omega(\log^6 N)$. Fletcher et al. [9] show how to decrease the number of interaction of Onion ORAM from $\log N$ to 1. Moataz et al. [21] 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 [21].

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 [31, 32]. Stefanov et al. [29] 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. [30] 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 [20, 26]. Recently, Garg et al. [10] improves the number of interactions of Path ORAM to be constant while inducing a multiplicative security overhead factor.

Due to the use of encryption, the security of all of the above related work is based on a computational hardness assumption. In contrast, this paper tackles information-theoretic security against unbounded adversary. Information-theoretic security is especially interesting in situations where encryption would overburden devices' computational capabilities, e.g., wireless sensors.

REFERENCES

[1] Andris Ambainis. Upper bound on communication complexity of private information retrieval. In *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, pages 401–407, 1997.

[2] 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.

[3] Josh Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In *Proceedings on Advances in cryptology*, pages 27–35. Springer-Verlag New York, Inc., 1990.

[4] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 41–50, 1995.

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

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

[7] 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.

[8] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally robust private information retrieval. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 269–283, 2012.

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 223–238, 1999.

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

[26] L. Ren, C.W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants Count: Prac-

tical Improvements to Oblivious RAM , 2014. IACR Cryptology ePrint Archive 997.

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

[28] 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.

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

[30] 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.

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

[32] 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.