Onion ORAM: A Constant Bandwidth and Constant Client Storage ORAM (without FHE or SWHE)

Srinivas Devadas†, Marten van Dijk‡, Christopher W. Fletcher†, Ling Ren†

† Massachusetts Institute of Technology – {devadas, cwfletch, renling}@mit.edu
‡ University of Connecticut – vandijk@engr.uconn.edu

January 18, 2015

Abstract

We present techniques to construct constant bandwidth, client storage and server storage blowup Oblivious RAM schemes in the (single-server) client-server setting. Crucially, our constructions *do not rely on Fully Homomorphic Encryption (FHE) or Somewhat Homomorphic Encryption (SWHE)* but instead rely only on efficient public-key additive homomorphic (additive-HE) encryption schemes such as the Damgård-Jurik cryptosystem (a generalization of the Paillier cryptosystem).

The key mechanism that we use to get constant bandwidth overhead is *client-guided server eviction*. To perform an ORAM eviction operation, the client transmits a *small* encrypted permutation matrix to the server, which the server combines with the additive-HE scheme and its locally-stored ORAM blocks to obliviously re-shuffle those blocks. By *small*, we mean that for appropriate parameter settings, the permutation matrix's amortized bandwidth cost is a constant with respect to the ORAM block size.

A serious problem with the above approach is that due to properties of additive-HE schemes, each block involved in a permutation gets an additional *layer of encryption*. Indeed, if combined with previous ORAM schemes naïvely, the number of encryption layers per-block grows unbounded (with the number of accesses made to the ORAM). This blows up server storage and bandwidth (due to ciphertext blowup) as well as client computation (as the client must "peel" off all layers to get the underlying plaintext). To address this challenge, we propose *Onion ORAM*, a new ORAM scheme that is designed and optimized to bound the number of encryption layers on each block to $\tilde{O}(\log N)$, where N is the number of blocks in the ORAM—*i.e., independent of the number of ORAM accesses*.

Putting it together, with sufficiently large block size $B = \Omega(k \log^2 N \log^2 \log N)$ bits for a security parameter k, Onion ORAM achieves $O(B)$ bandwidth, $O(B)$ client storage and $O(BN)$ server storage—only a constant factor blowup in all the three metrics. Using the Damgård-Jurik cryptosystem as our underlying primitive, Onion ORAM achieves the aforementioned asymptotics for block sizes $B = \Omega(\log^5 N \log^2 \log N)$ bits and security against known attacks with complexity $O\left(N^{\omega(1)}\right)$, super-polynomial in the security parameter.

1 Introduction

Oblivious RAM (ORAM), initially proposed by Goldreich and Ostrovsky [11, 12], is a cryptographic primitive that allows a client to store private data on an untrusted storage system (the server) and maintain *obliviousness* while accessing that data — i.e., guarantee that the server or any other observer learns nothing about the data and the client's access pattern (the sequence of addresses or operations) to that data. Since its initial proposal, ORAM has been studied in various application settings including cloud outsourced storage [26, 25, 6, 22, 18], secure processors [8, 17, 22, 7, 23, 21, 33] and secure multi-party computation [9, 10, 14, 16, 31]. We primarily consider the outsourced storage setting with a single server that is honest but curious (we do not consider malicious servers in this work).

1.1 Server Computation in ORAM

Historically, the ORAM model did no include server's ability to do computation, because the early works by Goldreich and Ostrovsky [11, 19, 13] considered secure hardware where the untrusted external memory is a storage device with no computation power. However, subsequent works [32, 25, 6] also investigated remote oblivious file servers, where the untrusted server usually has significant computational power (possibly even much more power than the client). Therefore, it is only natural to extend the ORAM model to include server computation as well.

Indeed, many recent ORAM schemes have implicitly or explicitly leveraged some amount of server computation to reduce bandwidth cost. For example, state of the art practical ORAM schemes such as the SSS construction [26], Oblivistore [25], Burst ORAM [6] and Ring ORAM [22] assumed the server is able to perform matrix multiplication or XOR operations. Path PIR [18] and subsequent work [34] increased allowed computation to additively homomorphic encryption such as Trostle-Parrish PIR [29] or Paillier's Cryptosystem [20]. Apon et al. [2], Path-PIR [18] and Gentry et al. [9, 10] further augmented ORAM with Fully Homomorphic Encryption (FHE). We remark that some prior works [18, 2] call themselves oblivious storage (or oblivious outsourced storage) to distinguish from the standard ORAM model where there is no server computation. We will simply refer to the two models as standard ORAM and ORAM with server computation only when it's not clear from context.

1.2 The Goldreich-Ostrovsky Lower Bound (and attempts to "break" it)

Traditionally, ORAM constructions are evaluated by their *bandwidth*, *client storage* and *server storage* Bandwidth is the amount of data that need to be sent between client/server to serve a client request, including the communication in the background to maintain the ORAM (i.e., ORAM evictions). Client storage is the amount of trusted local memory required at the client side to manage the ORAM protocol at any point in the protocol and server storage is the amount of storage needed at the server to store all data blocks.

In their seminal work [12], Goldreich and Ostrovsky showed that ORAM must incur a $O(\log N)$ lower bound in bandwidth blowup, for an ORAM of N blocks, under $O(1)$ blocks of client storage. If we allow the server to perform computation, however, the Goldreich-Ostrovsky lower bound no longer applies with respect to *client-server bandwidth*.[1]

Yet, it is still not easy to break the bound. For example, all existing schemes using non-FHE server computation only improve bandwidth by a constant factor relative to state of the art schemes without server computation [26, 6, 22, 18]. Using FHE as a primitive, Apon et al. [2] and Mayberry et al. [18] give an existence sketch for ORAM schemes that achieve optimal $O(1)$ bandwidth overhead, thereby breaking the lower bound assuming FHE. The problem with these latter schemes is their reliance on the heavyweight FHE primitive (in particular, both rely on FHE bootstrapping). Until FHE is becomes practical, its not clear that these schemes can be practical.

An open problem is whether we can break the Goldreich-Ostrovsky bound with server computation but without using FHE. And if so, what is the new lower bound on bandwidth and client storage? This is exactly the problem we consider in this paper.

1.3 Our Contribution and High-level Techniques

Specifically, we construct *Onion ORAM*, an ORAM with constant blowup in bandwidth, client storage and server storage, without the use of FHE [2, 18] or Somewhat Homomorphic Encryption [10] (SWHE). In particular, our scheme only relies on an additively homomorphic encryption scheme (e.g., the Damgård-Jurik cryptosystem [5] which is based on Paillier's cryptosystem [20]).

[1] We note that the Goldreich-Ostrovsky bound is in terms of the *number of operations* that must be performed. With server computation, the number of operations is still subject to the bound, but can be performed on the server-side which makes it possible to break the bound in terms of bandwidth between client and server. Since historically client-server bandwidth has been the most important metric for ORAM, we believe breaking the bound in terms of client-server bandwidth constitutes a significant advance.

Relying on only partially homomorphic schemes is appealing for several reasons. First, it means we can achieve optimal bandwidth for ORAM using very different assumptions than prior work — for example, that factoring is intractable or the decisional composite residuosity assumption if we use the Damgård-Jurik or Paillier's cryptosystem as our underlying primitive. Second, partially homomorphic encryption schemes are known to be relatively efficient in practice (e.g., homomorphic operations require a single multiplication or exponentiation) and simple to implement.

We now give an overview of the techniques in our new ORAM construction. Note that for the entire paper, we assume the outsourced storage setting for ORAM (i.e., not secure computation) with a single server that is honest but curious.

1.3.1 ORAM Evictions via *Block-Size Independent* Oblivious Shuffles

Our key idea is to turn ORAM evictions (which constitute a majority of bandwidth) into (mostly) non-interactive processes where the client "guides" the server to perform ORAM eviction operations by communicating a *block-size independent* encrypted permutation matrix to the server, which the server can then use to perform the eviction obliviously, using only additively homomorphic (additive-HE) operations (i.e., like a permutation operation in [1]). This step introduces (asymptotically) negligible bandwidth overhead if the block size is larger than the permutation matrix. It is also efficient in terms of computation. The number of data blocks the server computes on per eviction is poly-logarithmic in N, for an N record ORAM, and the underlying computation is modular exponentiation and multiplication operations over ciphertexts.

1.3.2 ORAM Design for Bounding and Refreshing the Number of Encryption Layers

Unfortunately, oblivious permutations that use just additive-HE operations have a serious limitation in ORAM settings: after each permutation, every data element involved in the permutation is wrapped in an additional *layer* of encryption [1]. Fundamentally, each permutation is made up of homomorphic select (or multiplexer) operations which require multiplication and addition. With an additive-HE scheme, we get addition for free and can emulate multiplication by treating one argument as a scalar. At a high level, we have the following interesting relations where $\otimes$ and $\oplus$ denote the underlying operations to implement scalar multiplication and addition:

$$\mathcal{E}(0) \otimes \mathcal{E}(m) = \mathcal{E}(0) \qquad \mathcal{E}(1) \otimes \mathcal{E}(m) = \mathcal{E}(\mathcal{E}(m))$$

Obliviously selecting one of m_1 and m_2 (m_2 in the following example) can be done by

$$\mathcal{E}(0) \otimes \mathcal{E}(m_1) \oplus \mathcal{E}(1) \otimes \mathcal{E}(m_2) = \mathcal{E}(0) \oplus \mathcal{E}(\mathcal{E}(m_2)) = \mathcal{E}(\mathcal{E}(m_2))$$

It is eazy to generalize the method to one-out-of-many selection. The problem is that after such an oblivious selection operation, the selected message (m_2 in the example) gets one one layer of encryption, and the client needs to decrypt twice (i.e., "peel off both layers").

Depending on the underlying cryptosystem, each layer adds ciphertext blowup and extra client computation (to peel off the layers). This is acceptable in voting and PIR settings [5, 18] because in those cases, the number of permutations over a common set of blocks is small. In ORAM, data is continuously shuffled and unless care is taken, the number of layers that accumulate on each block grows with $O(T)$ — where T is the number of ORAM accesses.

To draw an analogy, one can think of encryption layers in our setting as ciphertext noise in FHE schemes.[2] For this reason, applying FHE to ORAM has already been proposed [18, 2] and seems[3] straightforward: schemes can assume that the FHE bootstrapping operation is continually used to keep noise at reasonable

[2]This comparison is meant to be taken at a high level only. For instance, while noise and layers accumulate on blocks at the same points in the ORAM protocol, noise in FHE will eventually cause ciphertexts to become undecryptable. In principle, the client can always decrypt a noise-less ciphertext with layers given enough time.

[3]We stress "seems." Bootstrapping is known to be an expensive operation and it is unclear if schemes relying on continuous bootstrapping can be practical.

Table 1: **Our contribution.** B is the ORAM data block size in bits. N is the number of blocks and is polynomial in the security parameter. Recommended block size is the mimimum block size such that the asymptotical results in the table hold. Server and client computation is given by the *amount of data* that must be computed upon and makes no assumptions on the underlying cryptosystem.

Scheme	Recommended Block size	Overall Bandwidth	Computation		Storage	
			Client	Server	Client	Server
Path ORAM [28, 27]	$\Omega(\log^2 N)$	$O(B \log N)$	$O(B \log N)$	N/A	$O(B \log N)\omega(1)$	$O(BN)$
Circuit ORAM [30]	$\Omega(\log^2 N)$	$O(B \log N)\omega(1)$	$O(B \log N)\omega(1)$	N/A	$O(B)$	$O(BN)$
Path-PIR linear [18]	$\Omega(\log^5 N)$	$O(B \log N)\omega(1)$	$O(B \log N)$	$O(B \log^2 N)\omega(1)$	$O(B)$	$O(BN \log N)$
Onion ORAM	$\Omega(\log^5 N \log^2 \log N)$	$O(B)$	$O(B \log N)$	$O(B \log^2 N)\omega(1)$	$O(B)$	$O(BN)$

levels independent of T. With additive-HE schemes such as Paillier [20] or Damgård et al. [5], *there is no equivelent bootstrapping operation* and layers must be managed in some other way.

To address the encryption layer problem, we propose a new ORAM scheme called *Onion ORAM* whose eviction mechanism is designed and optimized to guarantee worst-case bounds on the number of layers that have accumulated on any block at any time. The main idea is to enable free "layer refresh" from guaranteed empty slots. We make the key observation that *if an ORAM algorithm guarantees that a particular slot in server memory is empty at a particular time, then without revealing any information about the underlying client data, that slot can be reset to zero layers at no computation cost.* Resetting encryption layers in this way is key to breaking the dependence between the number of layers on a slot and T. Unfortunately, prior ORAM schemes do not give sharp enough guarantees on slot emptiness [30, 28, 22]. Thus, we design an eviction procedure (using ideas from previous schemes) to additionally provide the empty bucket guarantee with overwhelming probability.

Using layer refresh through empty slots along with several other techniques, we prove that the number of layers that can accumulate on any block in Onion ORAM is bounded to $O(\log N)$ — *independent of the total number of ORAM accesses T.* (For the rest of the paper we assume logarithms are base 2.) We note that this bound makes no assumptions on the randomness in the ORAM protocol or the client access pattern.

Since bounding layers is related to controlling ciphertext noise (in the sense that both grow with circuit depth), our techniques will also be useful in a setting that uses FHE or SWHE as the underlying primitive. In particular, our scheme's circuit depth (in terms of two-input AND gates) for any block is also $O(\log N)$. Thus, a leveled FHE scheme (e.g., [4]) parameterized for this depth should be able to accomplish the entire protocol without the use of bootstrapping.

1.4 Technical Highlight: Optimal Asymptotics for Large Block Sizes

Combining the techniques from § 1.3 together, we show a parameterization for Onion ORAM in § 4.4 that yields $O(B)$ overall bandwidth, $O(B)$ client storage and $O(NB)$ server storage for relatively large block sizes $B = \Omega(\log^5 N \log^2 \log N)$ bits. Note that all three metrics are optimal, so this answers the question of the new lower bounds.

We summarize our results compared to prior-art schemes in Table 1. We add client and server computation in the table as two new metrics for ORAM and assume the Damgård-Jurik cryptosystem as the underlying primitive for both Onion ORAM and Path-PIR. All schemes use the standard recursion trick of [24], with small block sizes for the recursive ORAMs [27]. Following [28], for sufficiently large data block size $B = \Omega(\log^2 N)$, recursion does not change overall asymptotics. We remark that the Path PIR proposal [18] used a sub-optimal strategy for recursion, namely setting the same block size for each ORAM in the recursion. The entry in the table assumes the optimal recursion strategy for their scheme and adjusts their results accordingly. This is why the results in Table 1 for Path-PIR are better than what the original paper [18] reported. All the three schemes have at least $O(\log N)$ bandwidth blowup (even though Path PIR utilized server computation).

2 Formal Definitions

As described previously, the goal of ORAM is complete access pattern obfuscation: Formally, a standard ORAM without server computation can be defined as:

Definition 1. (ORAM Definition) *Let*

$$\overleftarrow{y} = ((\mathsf{op}_M, \mathsf{addr}_M, \mathsf{data}_M), \ldots, (\mathsf{op}_1, \mathsf{addr}_1, \mathsf{data}_1))$$

denote a RAM request sequence of length M, where op_i denotes whether the i-th operation is a read or write, addr_i denotes the address for that access and data_i denotes the data (if a write). Let $\mathsf{ORAM}(\overleftarrow{y})$ be the resulting randomized RAM request sequence. The ORAM protocol guarantees that for any $\overleftarrow{y}$ and $\overleftarrow{y}'$, $\mathsf{ORAM}(\overleftarrow{y})$ and $\mathsf{ORAM}(\overleftarrow{y}')$ are computationally indistinguishable if $|\overleftarrow{y}| = |\overleftarrow{y}'|$, and also that for any $\overleftarrow{y}$ the data returned to the client by the server is consistent with $\overleftarrow{y}$ (i.e., the ORAM behaves like a valid RAM) with overwhelming probability.

Extending this definition to handle our requirements on server computation is straightforward. We extend $\mathsf{ORAM}(\overleftarrow{y})$ from a randomized RAM request sequence, to *messages of arbitrary form* sent between the client and server to complete the protocol. This definition, also used in [18], is powerful in the sense that it makes no assumption on what types of additional information are passed between client and server yet clearly captures the essence of the original definition.

3 Basic Onion ORAM

We now present a basic version of Onion ORAM to illustrate the important features. We remark that the algorithm we describe in this section (Algorithm 1) is not the most competitive ORAM scheme under the standard ORAM definition (i.e., no server computation). Ring ORAM [22] and Circuit ORAM [30] beat it in different aspects. We present this algorithm because it will share the same framework as our final proposal (Section 4). Our final proposal will beat both Ring and Circuit ORAM in the server computation setting without using FHE.

We build on the binary tree ORAM framework of Shi et al. [24], which organizes server storage as a binary tree of nodes. The binary tree has $L + 1$ levels, where the root is at level 0 and the leaves are at level L. Each node in the binary tree is called a bucket and can contain up to Z data blocks. Each block is mapped to a random path in the binary tree via a position map. The position map may need to be recursively stored in other smaller ORAMs. When the data block size is $\Omega(\log^2 N)$ for an N element ORAM—which will be the case for all of our final parameterizations—the asymptotic costs of recursion are insignificant relative to other terms [27]. Thus, we will assume that the (recursed) position map adds negligible server storage and bandwidth for the rest of the paper.

Reading a block with address a in Algorithm 1 is similar to most tree-based ORAMs: look up the position map to obtain the path block a is currently mapped to, read all the blocks on that path to find block a, remap it to a new random path and add it to the root bucket (the root bucket and the *stash* from [28] are the same thing in our construction).

Our eviction strategy combines techniques from [24], [9] and [22]. For every A (a parameter proposed in [22], which we will set later) ORAM accesses, we select a path to evict based on the reverse lexicographical order of paths (proposed in [9]), denoted l_g in Algorithm 1. For the rest of the paper, paths (e.g., l_g) are represented as L-bit vectors where the i-th bit (from right to left) is set to 1 if the i-th bucket on the path goes to its right child.

To perform an eviction: For every bucket $\mathcal{P}(l_g, i)$ (i from 0 to L, i.e., from root to leaf) on path l_g, we move blocks from $\mathcal{P}(l_g, i)$ to its two children (similar to [24]). (It is not fundamental for the tree to be of degree 2, but we will assume this parameterization for the rest of the paper.) In each of these bucket-triplet evictions, we call $\mathcal{P}(l_g, i)$ the *source bucket*, the child bucket also on $\mathcal{P}(l_g)$ *destination bucket*, and the other child *sibling bucket*.

Algorithm 1 Basic Onion ORAM (no server computation).

function Access($a, \mathsf{op}, \mathsf{data}'$)
  Global/persistent variables: cnt **and** G**, initialized to** 0
  $l' \leftarrow$ UniformRandom$(0, 2^L - 1)$
  $l \leftarrow$ PositionMap$[a]$
  PositionMap$[a] \leftarrow l'$
  data $\leftarrow$ ReadPath(l, a)
  if op $=$ read **then**
    return data to client
  if op $=$ write **then**
    data $\leftarrow$ data$'$
  $\mathcal{P}(l, 0, \mathsf{cnt}) \leftarrow (a, l', \mathsf{data})$ ▷ add this block to the root
  cnt $\leftarrow$ cnt $+ 1 \mod A$
  if cnt $\stackrel{?}{=} 0$ **then**
    $l_g \leftarrow G \mod 2^L$ ▷ reverse lexicographical order
    EvictAlongPath(l_g)
    $G \leftarrow G + 1$

function ReadPath(l, a)
  Read all blocks on path $\mathcal{P}(l)$
  Select and return the block with address a
  Invalidate the block with address a
    ▷ Involves re-encrypting metadata of all the blocks on $\mathcal{P}(l)$ to hide which block is invalidated

function EvictAlongPath(l_g)
  for $i \leftarrow 0$ to $L - 1$ **do**
    Read all the blocks in $\mathcal{P}(l_g, i)$ and its two children
    Move all blocks in $\mathcal{P}(l_g, i)$ to its two children
    Write back the two children of $\mathcal{P}(l_g, i)$
    ▷ $\mathcal{P}(l_g, i)$ is gauranteed to be empty at this point (Observation 1)

A crucial change that we make to the eviction procedure of [24] is that we move *all* the blocks in the source bucket to its two children. This guarantees the following crucial invariant in our constructions in the next section.

Observation 1 (Empty Bucket Invariant)**.** *After a bucket-triplet eviction operation, the source bucket is empty.*

This observation immediately holds if the two children have enough room to accept all the incoming blocks from the source bucket. In other words, it holds if no bucket ever overflows. We guarantee this property by setting the bucket size Z and the eviction frequency A properly. According to the following theorem, if we simply set $Z = A = \Theta(\log N)\omega(1)$, the probability that a bucket overflows is $N^{-\omega(1)}$, negligible in security parameter (note that N is polynomial in security parameter).

Theorem 1. *If $Z \geq A$ and $N \leq A \cdot 2^{L-1}$, the probability that a bucket overflows after an eviction operation is bounded by $e^{-\frac{(2Z-A)^2}{6A}}$.*

Proof. First of all, notice that when $Z \geq A$, the root bucket will never overflow. So we will only consider non-root buckets. Let b be a non-root bucket, and $Y(b)$ be the number of blocks in it after an eviction operation. We will first assume all buckets have infinite capacity and show that $E[Y(b)] \leq A/2$, i.e., the

expected number of blocks in a non-root bucket after an eviction operation is no more than $A/2$ at any time. Then, we bound the overflow probabiliy given a finite capacity.

If b is a leaf bucket, each of the N blocks in the system has a probability of 2^{-L} to be mapped to b independently. Thus $E[Y(b)] \leq N \cdot 2^{-L} \leq A/2$.

If b is a non-leaf (and non-root) bucket, we define two variables m_1 and m_2: the last EvictAlongPath operation where b is on the eviction path is the m_1-th EvictAlongPath operation, and the EvictAlongPath operation where b is a sibling bucket is the m_2-th EvictAlongPath operation. If $m_1 > m_2$, then $Y(b) = 0$, because b becomes empty when it is the source bucket in the m_1-th EvictAlongPath operation. (Recall that buckets have infinite capacity so this outcome is guaranteed.) If $m_1 < m_2$, there will be some blocks in b and we now analyze what blocks will end up in b. We time-stamp the blocks as follows. When a block is accessed and remapped, it gets time stamp m^*, which is the number of EvictAlongPath operations that have happened. Blocks with $m^* \leq m_1$ will not be in b as they will go to either the left child or the right child of b. Blocks with $m^* > m_2$ will not be in b as the last eviction operation that touches b (m_2-th) has already passed. Therefore, only blocks with time stamp $m_1 < m^* \leq m_2$ can be in b. There are at most $d = A|m_1 - m_2|$ such blocks. Such a block goes to b if and only if it is mapped to a path containing b. Thus, each block goes to b independently with a probability of 2^{-i}, where i is the level of b. The deterministic order of EvictAlongPath makes it easy to see[4] that $|m_1 - m_2| = 2^{i-1}$. Therefore, $E[Y(b)] \leq d \cdot 2^{-i} = A/2$ for any non-leaf bucket as well.

Now that we have independence and the bound on expectation, a simple Chernoff bound completes the proof. □

From the empty bucket invariant (Observation 1), we immediately get another two observations, which will be useful in our final construction.

Observation 2. *After* EvictAlongPath(l_g)*, all the non-leaf buckets on* $\mathcal{P}(l_g)$ *are empty.*

Observation 3. *At the beginning of* EvictAlongPath*, all the non-leaf sibling buckets of* $\mathcal{P}(l_g)$ *are empty.*

Observation 2 holds simply because every non-leaf bucket on $\mathcal{P}(l_g)$ has just been involved in a bucket-triplet eviction as the source bucket, which would become empty due to Observation 1. Note that due to the reverse lexicographical eviction order, any sibling bucket on this eviction must be on the eviction path the last time it was involved in an eviction. Then, Observation 3 directly follows from Observation 2.

Metadata. Many operations in Algorithm 1 are implemented with the help of metadata—its address and leaf label (the path the block is mapped to)—stored alongside each data block (see Line 11, Algorithm 1). Selecting the block of interest (Line 19) is done by comparing the address of every block on the path with the address of interest a. Invalidating the block of interest (Line 20) means changing its address field to $\perp$, an address reserved for dummy blocks. This requires re-encrypting the address field of every block on the path to hide the block being invalidated. In eviction, whether a block goes to the destination bucket or the sibling bucket depends on its leaf label.

Asymptotics without server computation. We conclude this section with a summary of the asymptotic performance of this ORAM construction. Note that the parameters satisfy $Z = A = O(\log N)\omega(1)$ and $N \leq A \cdot 2^{L-1}$. Reading a block involves downloading all the $Z(L+1) = O(\log^2 N)\omega(1)$ blocks on a path. Eviction also touches $O(ZL) = O(\log^2 N)\omega(1)$ blocks. So the amortized and worst case bandwidth are both $O(\log^2 N)\omega(1)$ blocks. Client storage is $O(Z) = O(\log N)\omega(1)$ blocks; server storage is $O(2^{L+1} \cdot Z) = O(N)$ blocks.

4 Homomorphic Onion ORAM

We now describe the cryptographic primitives we need and the final Onion ORAM scheme.

[4]One way to see this is that a bucket b at level i will be on the evicted path every 2^i EvictAlongPath operations, and its sibling will be on the evicted path halfway in that period.

4.1 Primitive Operations

4.1.1 General Notation

We denote a plaintext space or ciphertext space as $\mathbb{L}_i$. $|\mathbb{L}_i|$ is the number of bits needed to represent an element in $\mathbb{L}_i$. $\mathbb{L}_i^a$ is the space of length a vectors in $\mathbb{L}_i$. Vectors are bolded; if $\mathbf{V}$ is a vector, $\mathbf{V}[i]$ is the i-th element.

4.1.2 Requirements on Underlying Cryptosystem

We require a series of additive homomorphic cryptosystems, given as the triplets $\mathsf{AHE}_i = (\mathcal{G}_i, \mathcal{E}_{\mathsf{pk}_i}, \mathcal{D}_{\mathsf{sk}_i})$ for $i \geq 0$ where $\mathsf{sk}_i, \mathsf{pk}_i = \mathcal{G}_i()$, and the ciphertext space of AHE_i is in the plaintext space of AHE_{i+1}. We denote the plaintext space of AHE_i as $\mathbb{L}_i$. The homomorphism of AHE_i ($i \geq 0$) has properties:

1. $\mathcal{E}_i : \mathbb{L}_i \to \mathbb{L}_{i+1}$
2. $\mathcal{D}_i : \mathbb{L}_{i+1} \to \mathbb{L}_i$
3. Homomorphic Addition $\oplus$: $\mathbb{L}_{i+1} \times \mathbb{L}_{i+1} \to \mathbb{L}_{i+1}$, $\mathcal{E}_i(x) \oplus \mathcal{E}_i(y) = \mathcal{E}_i(x + y)$
4. ...calar Multiplication $\otimes$: $\mathbb{L}_{i+1} \times \mathbb{L}_i \to \mathbb{L}_{i+1}$, $\mathcal{E}_i(x) \otimes y = \mathcal{E}(x \cdot y)$

...re $\oplus$ and $\otimes$ are efficiently computable functions and x and y are valid plaintexts. Additionally, we require ...at each AHE_i has a valid zero element and a valid identity element. Formally:

ie) (3.)]TJΩ0 g 0 GΩ [-500on19.648 -1.49f8 9.9626 Tf 3.317 1.002

5. $\mathcal{E}_i(e)$ is efficiently p18.431 Td [(3.)]AHEΩ0 g 0 GΩ [-500on19.648 -1.494 Td [(i)]TJ/F8 9.9626 Tf 6.638 1.494 Td [(has)-33

4.959.79]TJ/F8 9.9626 Tf 4.639 0 Td [())-35938 0 Td [(V)]TJ/F8 9.9626 Tf 11.

4.2 Homomorphic Onion ORAM (Protocol)

We now describe how to take advantage of server computation and our above primitives to improve the basic Onion ORAM construction from § 3. The high level idea is as follows. Recall from § 3 that most operations in Onion ORAM rely on metadata. To perform a ReadPath or EvictAlongPath, the client will still download *and manage* metadata and use it to construct choice vectors or permutation matrices required by our additive-HE Select operation. The server uses a single (or small number of) additive-HE Select operations on data blocks to return the block of interest and uses a poly-logarithmic number of Select operations to carry out evictions. In this way, the client and the server only exchange one block plus choice vector per ORAM request and no blocks (up to a constant number of blocks) plus poly-logarithmic number of choice vectors per eviction. We refer to the choice vectors involved in the eviction as the permutation matrix. When the block size is sufficiently large and dominates the metadata and permutation matrix, we get constant bandwidth blowup.

4.2.1 Simplifying Assumptions and Additional Parameters

We introduce some more notations for later use. For a path $\mathcal{P}(l)$ in the ORAM tree, $\mathcal{P}(l).\mathsf{address}$ is a vector of length $Z(L+1)$ consisting of the address fields of all the blocks on $\mathcal{P}(l)$, from the root to the leaf. $\mathcal{P}(l).\mathsf{leaf}$ and $\mathcal{P}(l).\mathsf{data}$ are vectors of length $Z(L+1)$ consisting of leaf labels and data of all the blocks on $\mathcal{P}(l)$, respectively. Similarly, for a bucket $\mathcal{P}(l,k)$, we define $\mathcal{P}(l,k).\mathsf{address}$, $\mathcal{P}(l,k).\mathsf{leaf}$ and $\mathcal{P}(l,k).\mathsf{data}$ to be vectors of length Z, consisting of address, leaf label and data fields of all the blocks in that bucket, respectively. All the data fields will be homomorphically evaluated by the server, and thus be encrypted (with layers) using the public-key additive-HE scheme. All the metadata are still managed by the client, and can use any semantically secure symmetric encryption, e.g., AES counter mode.

To simplify the basic version of our protocol, we add two simple rules which will be optimized in later sections. First, for now we assume each Select operation is implemented as the flat circuit from § 4.1.3. That is, a Select operation that chooses between b data blocks requires a choice vector π consisting of b coefficients.

Parameter i_{max}. Second, for each $\mathsf{Select}(\pi, \mathbf{V})$ operation, we assume both the server and the client know the number of layers on each input block involved. This assumption is explained in detail in § 4.3. For each Select operation, we designate the maximum number of layers among the input blocks as i_{max}. Then, immediately before a Select operation, the server will 'promote' every input block to i_{max} layers by running the encryption routine $\mathcal{E}$ on that block until it is in $\mathbb{L}_{i_{max}}$. This way, the server forms the input $\mathbf{V} \in \mathbb{L}^a_{i_{max}}$ for $|\mathbf{V}| = a$. We assume the client also knows i_{max} (e.g., from metadata), and sends $\pi \in \mathbb{L}_{i_{max}+1}$.

Parameter I_{max}. We also add a single public parameter called I_{max}. Conceptually, I_{max} is the maximum number of encryption layers the client is willing to let accumulate on any block. In § 4.3, we prove that $I_{max} = \Theta(\log N)$ is a reasonable setting for this parameter.

4.2.2 Initialization

To initialize, the client runs $\mathcal{G}$ and sends pk to the server.

4.2.3 Read Path

$\mathsf{ReadPath}(l, a)$ can be done by a simple Select operation. We note that this is similar to how reads are performed in [18], except that in our case the block of interest may be encrypted under many layers. Below are the steps in detail. The client first looks up the position map to determine the path l to read. Then:

1. Client to server: l
2. Server to client: all the addresses on that path, namely $\mathcal{P}(l).\mathsf{address}$
3. Client computation: the client decrypts $\mathcal{P}(l).\mathsf{address}$ and finds j such that $\mathcal{P}(l).\mathsf{address}[j] = a$ (i.e., the index of the block of interest). It constructs the vector π of length $Z(L+1)$ where $\pi[j] = \mathcal{E}_{i_{max}}(1)$ and other components are $\mathcal{E}_{i_{max}}(0)$ where i_{max} is defined in § 4.2.1.

4. Client to server: π
5. Server computation: the server 'promotes' every element in $\mathcal{P}(l).\mathsf{data}$ to $\mathbb{L}_{i_{max}}$, forming $\mathbf{V}$ as described in § 4.2.1, and then evaluates $d = \mathsf{Select}(\pi, \mathbf{V})$.
6. Server to client: d
7. Client computation: the client gets the plaintext data m by running the decryption routine $i_{max} + 1$ times, namely $m = \mathcal{D}^{i_{max}+1}(d)$. If the client operation is a write, the client updates the data forming a new block m'.
8. Client to server (the invalidation step): the client sets $\mathcal{P}(l).\mathsf{address}[\mathsf{j}] = \perp$, re-encrypts $\mathcal{P}(l).\mathsf{address}$ and sends it to the server. The server updates $\mathcal{P}(l).\mathsf{address}$.
9. Client computation: the client encrypts the message to get $d' = \mathcal{E}_0(m')$.
10. Client to server: d'
11. The server copies d' to a public offset in the root bucket $\mathcal{P}(l, 0, \mathsf{cnt})$ as described in Algorithm 1.

As we show in § 4.3, the server can determine which buckets along $\mathcal{P}(l)$ are empty at the start of any $\mathsf{ReadPath}()$ operation. Thus, it is sufficient to only transfer metadata/encrypted coefficients and perform Select operations for non-empty buckets in the above algorithm, which saves a constant factor bandwidth.

4.2.4 Evict Along Path

At any time, both the client and the server know which path to evict along by the deterministic eviction order. Let the path be l_g. For $\mathsf{EvictAlongPath}(l_g)$, each bucket-triplet eviction operation can be done by $O(Z)$ Select operations.

We point out the important property that all non-leaf sibling buckets of $\mathcal{P}(l_g)$ are guaranteed to be empty at this point due to Observation 3. Leveraging this property, we will show a smart way to move blocks from a source bucket to a sibling bucket without even invoking a Select operation.

For each level k from 0 to $L - 1$, the client and server perform the following steps.

1. Server to client: all the addresses and leaf labels for the source bucket and destination bucket, namely $\mathcal{P}(l_g, k).\mathsf{address}$, $\mathcal{P}(l_g, k).\mathsf{leaf}$, $\mathcal{P}(l_g, k+1).\mathsf{address}$ and $\mathcal{P}(l_g, k+1).\mathsf{leaf}$. As pointed out, the sibling bucket is guaranteed to be empty at this point. Thus, its metadata need not be transferred.
2. Client computation: the client computes the bucket-triplet eviction as before, i.e., moves all the blocks in the source bucket to either the destination bucket or the sibling bucket. The client generates new metadata (addresses and leaf labels) for all the three buckets.
3. (Source to sibling eviction) Since the sibling bucket is currently empty, the server can simply copy the source bucket into the sibling bucket, and the client can invalidate the blocks that should not move into the sibilng bucket. In more detail,
   (a) The server copies $\mathcal{P}(l, k).\mathsf{data}$ and $\mathcal{P}(l, k).\mathsf{leaf}$ from the source bucket into the sibling bucket.
   (b) The client generates the new address field for the sibling bucket, by invalidating (setting address to $\perp$) the blocks that do not move into the sibling bucket. The client re-encrypts the new address field and sends it back to the server where the server writes it to the sibling bucket.

   After this operation, the number of layers in the sibling bucket is the same as the original number of layers in the source bucket (as opposed to adding one more layer).

4. (Source to destination eviction) For this step, the client needs to guide the server to perform $O(Z)$ Select operations. Note that this step does not fully permute blocks from source to destination: we only need to move all the blocks in the source bucket to the destination bucket, while leaving blocks already in the destination bucket unchanged. For each bucket slot z from 0 to $Z-1$, the client and server perform the following steps.

   (a) Client computation: Suppose $\mathbf{V}$ is the concatenation of the source bucket and the z-th slot of the destination bucket. The client generates the choice vector π, where $\pi[j] = \mathcal{E}_{i_{max}}(1)$ if and only if the j-th block in $\mathbf{V}$ moves to the z-th slot in the detination bucket (and $\pi[j] = \mathcal{E}_{i_{max}}(0)$ otherwise).

   (b) Client to server: π

   (c) Server computation: the server 'promotes' $\mathcal{P}(l_g, k).\mathsf{data}$ and $\mathcal{P}(l_g, k+1, z).\mathsf{data}$ to $\mathbb{L}_{i_{max}}$, forming $\mathbf{V}$, and then updates the z-th slot in the destination bucket to $\mathsf{Select}(\pi, \mathbf{V})$.

   (d) Client to server: The client sends the updated metadata for the source bucket and the destination bucket, (i.e., $\mathcal{P}(l_g, k).\mathsf{address}$, $\mathcal{P}(l_g, k).\mathsf{leaf}$, $\mathcal{P}(l_g, k+1).\mathsf{address}$ and $\mathcal{P}(l_g, k+1).\mathsf{leaf}$) and the server updates them.

Eviction Post-Processing. The careful reader will notice that once blocks arrive at leaf buckets during evictions, they will accumulate layers in the leaves until they are requested by the client. To decouple the number of layers on blocks from the client access pattern, we perform the following post-processing step on the destination leaf bucket $\mathcal{P}(l_g, L)$ at the end of each $\mathsf{EvictAlongPath}()$ operation. For each bucket slot z from 0 to $Z-1$, the client and server perform the following steps.

1. Server to client: $d = \mathcal{P}(l_g, L, z).\mathsf{data}$
2. Client computation: runs the decryption or encryption routine on d until the ciphertext is in $\mathbb{L}_{I_{max}-1}$, giving us d'. Then the client runs the encryption routine one additional time, forming $d'' = \mathcal{E}_{I_{max}-1}(d')$.
3. Client to server: d''
4. The server then updates memory as $\mathcal{P}(l_g, L, z).\mathsf{data} = d''$.

This step ensures that blocks in the leaf buckets don't accumulate more than I_{max} layers. By setting I_{max} to equal the worst-case bound on layers (§ 4.3), this step further ensures that no block at any point accumulates more than I_{max} layers. We make two remarks related to metrics. First, this step may be performed in a single or multiple roundtrips, depending on the client storage requirements. Seond, if $Z \sim A$ as we assume from § 3, the bandwidth overhead of this step is constant.

4.3 Bounding the Layers

We now prove that the number of layers that can accumulate on blocks by the time they arrive at leaf buckets is bounded to $O(\log N)$. Combining this analysis with the eviction post-processing step (§ 4.2.4) gives us a bound for all blocks at all points in the protocol.

Theorem 2. *For a bucket at level $k \leq L$, the number of layers on that bucket is bounded by $k+1$.*

To prove the above theorem, we first point out that how a bucket transitions between the three roles (source, destination and sibling) in bucket-triplet evictions. The root bucket is always the source bucket, and the leaves are never source buckets; these are two special cases. For the general case (non-root, non-leaf buckets), we have the following lemma.

Lemma 1. *A bucket at level k where $0 < k < L$ (i.e., non-root and non-leaf) transitions between destination bucket, source bucket and sibling bucket, in that order.*

Proof. This is easy to see from our EvictionAlonePath() operation and the reverse lexicographic order of eviction paths. When a bucket b is on the eviction path, it is first involved in a bucket-triplet eviction as the destination bucket (when its parent is the source bucket), and then immediately as the source bucket itself in the very next bucket-triplet eviction. Then, due to the reverse lexicographic order, the sibling of bucket b will be on the eviction path, before the eviction path loops back through b again. When b's sibling is on the eviction path, b will be involved in a bucket-triplet eviction as the sibling bucket. Due to the reverse lexicographic order again, after that b will be on the eviction path before its sibling, and a new period starts for b. □

We introduce the notation (role, level, layers). For example, (source, k, j) means a bucket at level k currently has *at most* j layers, and will be the source bucket when it is next involved in a bucket-triplet eviction. With the role transition pattern in Lemma 1, we can prove the following theorem with induction.

Theorem 3. *A bucket at level k where $k < L$ (i.e., non-leaf) is always in one of the following three states:* (source, k, $k+1$), (destination, k, k), (sibling, k, 0).

Proof. Initially, the theorem trivially holds because all buckets have zero layers. Now suppose every bucket is in one of the three allowed states at some point. We prove that after a bucket-triplet eviction, the three buckets involved will still be in an allowed state (all the other buckets in the tree are not affected and remain in their good states).

Let us now focus on a bucket-triplet eviction where the source bucket is at level k. Then, the current states of the three involved buckets are, by definition and by the induction hypothesis (source, k, $k+1$), (destination, $k+1$, $k+1$) and (sibling, $k+1$, 0). We will analyze the three involved buckets one by one.

The current source bucket will become empty, and be reset to zero layers. If it is the root ($k = 0$), it will always be a source bucket. Otherwise it will transition into a sibling bucket by Lemma 1. So its new state is either (source, 0, 0) or (sibling, k, 0), both of which are allowed states.

The current destination bucket will acquire one more layer on top of the maximum number of layers between the current source and current destination bucket (itself), both of which are bounded by $k+1$. Then, if $k+1 < L$ the current destination bucket will transition into a source bucket by Lemma 1, importantly at level $k+1$. So its new state is (source, $k+1$, $k+2$), which is an allowed state. (If $k+1 = L$, it is a leaf bucket and this theorem does not care about it.)

The current sibling bucket will be copied from the source bucket, and thus will have at most $k+1$ layers. If $k+1 < L$, it will transition into a destination bucket by Lemma 1. So its new state is (destination, $k+1$, $k+1$), which is an allowed state. (If $k+1 = L$, it is a leaf bucket and this theorem does not care about it.) □

Proof of Theorem 2. For non-leaf buckets, the proof directly follows from Theorem 3. With the post-processing step on the leaf buckets from § 4.2.4, we can bound the layers on leaf buckets similarly by induction. First, we set $I_{max} = L$. Leaf buckets transition between destination buckets and sibling buckets, and the two allowed states are (destination, L, $L+1$) and (sibling, L, L). (destination, L, $L+1$) gets one more layer, but is refreshed back to L layers, so it transitions into (sibling, L, L). (sibling, L, L) gets one more layer and transitions into (destination, L, $L+1$). □

Observation 4. *At any point, the server can determine the current number of layers on any bucket.*

Setting i_{max} (§ 4.2.1). By the above analysis, it is sufficient to set $i_{max} = L+1$ for a ReadPath() operation and $i_{max} = k+1$ in EvictAlongPath() for a bucket-triplet eviction whose source bucket is level k. By Observation 4, the server knows how to 'promote' blocks before a Select operation.

4.4 Parameterization for Desired Asymptotics

We now analyze the requirements on block size B for our Onion ORAM to get constant bandwidth blowup. To be concrete, we assume our underlying cryptosystem is the Damgård-Jurik cryptosystem [5] which we now summarize in the context of our requirements from § 4.1.2.

4.4.1 Background: Damgård-Jurik Cryptosystem

The Damgård-Jurik cryptosystem, a generalization of Paillier's cryptosystem [20], is based on the hardness of the decisional composite residuosity assumption. In this system, the public key $\mathsf{pk} = n = pq$ is an RSA modulus (p and q are two large, random primes) and the secret key $\mathsf{sk} = \mathsf{lcm}(p-1, q-1)$. In the terminology from § 4.1.2, $\mathsf{sk}, \mathsf{pk} = \mathcal{G}_i()$ for $i \geq 0$.

We denote the integers mod n as $\mathbb{Z}_n$. The message space for the i-th layer of the Damgård-Jurik cryptosystem encryption, $\mathbb{L}_i$, is $\mathbb{Z}_{n^{s_0+i}}$ for some user specified choice of s_0. The ciphertext space for this layer is $\mathbb{Z}_{n^{s_0+i+1}}$. Thus, we clearly have the property that ciphertexts are valid plaintexts in the next layer. An interesting property that immediately follows is that if $s_0 = \Theta(i)$, then $|\mathbb{L}_i|/|\mathbb{L}_0|$ is a constant. In other words, by setting s_0 appropriately the ciphertext blowup after i layers of encryption is a constant.

We further have that $\oplus$ (the primitive for homomorphic addition) is integer multiplication and $\otimes$ (for scalar multiplication) is modular exponentiation. If these operations are performed on ciphertexts in $\mathbb{L}_i$, operations are mod $\mathbb{Z}_{n^{s_0+i}}$.

4.4.2 Onion ORAM Parameters

Suppose we divide each plaintext block into C chunks, each chunk having B_c bits (i.e., $C \cdot B_c = B$). In the Damgård-Jurik cryptosystem (§ 4.4.1), each plaintext chunk can be represented by a number in $\mathbb{Z}_{n^{s_0}}$ for a user choice of s_0, which gives each message $s_0|n|$ bits. Due to Theorem 2, a block will accumulate at most $L + 1 = O(\log N)$ layers (if we use flat muxes). So we can set $s_0 = \Theta(\log N)$, and guarantee a constant ciphertext blowup. This means $B_c = \Theta(|n| \log N)$.

Second, we will set parameters so that the amortized bandwidth cost to send the encrypted permutation matrix for $\mathsf{EvictAlongPath}$ does not exceed $O(B)$ bits.[5] We will assume $Z = A = O(\log N)\omega(1)$ as described in § 3. Amortized over A and using the flat mux Select implementation, the client needs to send $\omega(\log^2 N)$ encrypted coefficients for the permutation matrix. Each coefficient is a ciphertext of $\Theta(|n| \log N)$ bits. Therefore, we want $\omega(\log^2 N)\Theta(|n| \log N) \sim O(B)$, or equivalantly $B = \omega(|n| \log^3 N)$.

If we apply the tree mux optimization from § 4.1.3, the bound on the number of layers increases to $O(\log N \log\log N)$, requiring $s_0 = \Theta(\log N \log\log N)$. At the same time, the amortized permutation matrix size reduces to $\Theta(\log N \log\log N)$ coefficients, each being a ciphertext of $\Theta(|n| \log N \log\log N)$ bits. Thus, the block size requirement becomes $\Theta(\log N \log\log N)\Theta(|n| \log N \log\log N) \sim O(B)$, or equivalently $B = \Omega(|n| \log^2 N \log^2 \log N)$.

Given n, the decisional composite residuosity assumption can be solved in time $\mathsf{exp}(|n|^{1/3} \log^{2/3} |n|)$ by using the general number field sieve to factor n. This is also the best known attack for the Damgård-Jurik cryptosystem. It therefore suffices to set $|n| = \Theta(\log^3 N)$, making the above attack superpolynomial (i.e., of complexity $N^{\omega(1)}$ following prior ORAM works [28, 30]). This setting gives the asymptotics in Table 1 at the beginning of the paper, i.e., a block size of $B = \Omega(\log^5 N \log^2 \log N)$ bits using tree muxes. We note that for practical parameterizations, previous ORAM works assume a security parameter of 80. To achieve security against attacks of complexity 2^{80}, $|n| = 1024$ bits is a reasonable setting [3].

5 Conclusion and Future Work

This paper proposes *Onion ORAM*, the first ORAM scheme with optimal asymptotics in bandwidth, server storage and client storage in the single-server setting. Critically, our construction does not require the use of Fully Homomorphic Encryption (FHE) or Somewhat Homomorphic Encryption (SWHE) and instead only requires a partially (additive) homomorphic scheme such as the Damgård-Jurik cryptosystem (based on Paillier's classical scheme). Due to the known efficiency of these types of cryptosystems, we think of our work as a first step towards *practical* constant bandwidth blowup ORAM schemes.

We leave several questions open as future work. First, without compromising asymptotics for bandwidth or client/server storage, what is the lower bound on block size? At present, our scheme requires a relatively

[5] Recall the permutation matrix consists of the Z choice vectors $\pi_0 \ldots \pi_{Z-1}$ from Step 4 in § 4.2.4.

large block size, e.g., $\Omega(\log^5 N \log^2 \log N)$ bits. For comparison, state of the art ORAM schemes that don't use homomorphic operations can achieve their best asymptotics with a block size of $\Omega(\log^2 N)$ bits. Second, independent of block size, what are lower bounds on *client* and *server computation*? Our scheme requires $O(B \log N)$ client computation and $O(B \log^2 N)\omega(1)$ server computation. While this amount of client computation is on-par with prior schemes, the Goldreich-Ostrovsky bound gives a $O(B \log N)$ bound on server computation, which suggests there is room for improvement.

6 Acknowledgements

We thank Elaine Shi for helpful feedback on early versions of this manuscript.

References

[1] B. Adida and D. Wikström. How to shuffle in public. Cryptology ePrint Archive, Report 2005/394, 2005. http://eprint.iacr.org/.

[2] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In *Public-Key Cryptography–PKC 2014*, pages 131–148. Springer, 2014.

[3] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for key management part 1: General (revision 3). *NIST Special Publication 800-57*, 2012.

[4] Z. Brakerski, G. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012*, pages 309–325, 2012.

[5] I. Damgard and M. Jurik. A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography*, pages 119–136, 2001.

[6] J. Dautrich, E. Stefanov, and E. Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, San Diego, CA, Aug. 2014. USENIX Association.

[7] C. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *Proceedings of the Int'l Symposium On High Performance Computer Architecture*, 2014.

[8] C. Fletcher, M. van Dijk, and S. Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing; an extended version is located at http://csg.csail.mit.edu/pubs/memos/Memo508/memo508.pdf (Master's thesis)*, pages 3–8, Oct. 2012.

[9] 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 *Privacy Enhancing Technologies (PET)*, 2013.

[10] C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345, 2014.

[11] O. Goldreich. Towards a theory of software protection and simulation on oblivious rams. In *STOC*, 1987.

[12] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[13] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. In *J. ACM*, 1996.

[14] M. Keller and P. Scholl. Efficient, oblivious data structures for mpc. Cryptology ePrint Archive, Report 2014/137, 2014. http://eprint.iacr.org/.

[15] H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *ISC*, pages 314–328, 2005.

[16] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient ram-model secure computation. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 623–638, Washington, DC, USA, 2014. IEEE Computer Society.

[17] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. ACM CCS, 2013.

[18] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining oram and pir. In *Proceedings of NDSS*, 2014.

[19] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.

[20] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Eurocrypt*, pages 223–238, 1999.

[21] L. Ren, C. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious-ram. In *Proceedings of the 17th IEEE High Performance Extreme Computing Conference*, September 2013.

[22] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. V. Dijk, and S. Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. Cryptology ePrint Archive, Report 2014/997, 2014. http://eprint.iacr.org/.

[23] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the Int'l Symposium on Computer Architecture*, June 2013. Available at Cryptology ePrint Archive, Report 2013/76.

[24] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt*, pages 197–214, 2011.

[25] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[26] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[27] E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. ACM CCS, 2013. Available at Cryptology ePrint Archive, Report 2013/280.

[28] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the ACM Computer and Communication Security Conference*, 2013.

[29] J. Trostle and A. Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 114–128, Berlin, Heidelberg, 2011. Springer-Verlag.

[30] X. S. Wang, T.-H. H. Chan, and E. Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. Cryptology ePrint Archive, Report 2014/672, 2014. http://eprint.iacr.org/.

[31] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 191–202, New York, NY, USA, 2014. ACM.

[32] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[33] X. Yu, C. W. Fletcher, L. Ren, M. van Dijk, and S. Devadas. Generalized external interaction with tamper-resistant hardware with bounded information leakage. In *Proceedings of the Cloud Computing Security Workshop (CCSW)*, 2013.

[34] J. Zhang, Q. Ma, W. Zhang, and D. Qiao. Kt-oram: A bandwidth-efficient oram built on k-ary tree of pir nodes. Cryptology ePrint Archive, Report 2014/624, 2014. http://eprint.iacr.org/.