Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM

Srinivas Devadas[†], Marten van Dijk[‡], Christopher W. Fletcher[†*], Ling Ren[†*], Elaine Shi[⋈], Daniel Wichs[∘]

† Massachusetts Institute of Technology – {devadas, cwfletch, renling}@mit.edu
‡ University of Connecticut – vandijk@engr.uconn.edu
⋈ Cornell University – elaine@cs.cornell.edu
∘ Northeastern University – wichs@ccs.neu.edu
∗ Lead authors

No Institute Given

Abstract. We present Onion ORAM, an Oblivious RAM (ORAM) with constant worst-case bandwidth blowup that leverages poly-logarithmic server computation to circumvent the logarithmic lower bound on ORAM bandwidth blowup. Our construction does not require fully homomorphic encryption, but employs an additively homomorphic encryption scheme such as the Damgård-Jurik cryptosystem, or alternatively a BGV-style somewhat homomorphic encryption scheme without bootstrapping. At the core of our construction is an ORAM scheme that has "shallow circuit depth" over the entire history of ORAM accesses. We also propose novel techniques to achieve security against a malicious server, without resorting to expensive and non-standard techniques such as SNARKs. To the best of our knowledge, Onion ORAM is the first concrete instantiation of a constant bandwidth blowup ORAM under standard assumptions (even for the semi-honest setting).

1 Introduction

Oblivious RAM (ORAM), initially proposed by Goldreich and Ostrovsky [19, 20, 36], is a cryptographic primitive that allows a *client* to store private data on an *untrusted server* and maintain *obliviousness* while accessing that data — i.e., guarantee that the server or any other observer learns nothing about the data or the client's access pattern (the sequence of addresses or operations) to that data. Since its initial proposal, ORAM has been studied in theory [21, 25, 39, 41, 45, 49], or in various application settings including secure outsourced storage [8, 29, 32, 42, 43, 50], secure processors [10–12, 31, 38, 40, 51] and secure multi-party computation [13, 14, 24, 28, 47, 48].

1.1 Server Computation in ORAM

The ORAM model considered historically, starting with the work of Goldreich and Ostrovsky [19, 20, 36], assumed that the server acts as a simple storage device

that allows the client to read and write data to it, but does not perform any computation otherwise. However, in many scenarios investigated by subsequent works [8,32,42,50] (e.g., the setting of remote oblivious file servers), the untrusted server has significant computational power, possibly even much greater than that of the client. Therefore, it is natural to extend the ORAM model to allow for server computation, and to distinguish between the amount of computation performed by the server and the amount of communication with the client.

Indeed, many recent ORAM schemes have implicitly or explicitly leveraged some amount of server computation to either reduce bandwidth cost [1,7,13,14, 29,32,39,43,52], or reduce the number of online roundtrips [49]. We remark that some prior works [1,32] call themselves oblivious storage (or oblivious outsourced storage) to distinguish from the standard ORAM model where there is no server computation. We will simply apply the term ORAM to both models, and refer to ORAM *with/without server computation* to distinguish between the two.

At first, many works implicitly used server computation in ORAM constructions [13,14,32,39,43,49,52], without making a clear definitional distinction from standard ORAM. Apon et al. were the first to observe that such a distinction is warranted [1], not only for the extra rigor, but also because the definition renders the important Goldreich-Ostrovsky ORAM lower bound [20] inapplicable to the server computation setting — as we discuss below.

1.2 Attempts to "Break" the Goldreich-Ostrovsky Lower Bound

Traditionally, ORAM constructions are evaluated by their *bandwidth*, *client storage* and *server storage*. Bandwidth is the amount of communication (in bits) between client/server to serve a client request, including the communication in the background to maintain the ORAM (i.e., ORAM evictions). We also define bandwidth blowup to be bandwidth measured in the number of blocks (i.e., blowup compared to a normal RAM). Client storage is the amount of trusted local memory required at the client side to manage the ORAM protocol and server storage is the amount of storage needed at the server to store all data blocks.

In their seminal work [20], Goldreich and Ostrovsky showed that an ORAM of N blocks must incur a $O(\log N)$ lower bound in bandwidth blowup, 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]. The reason is that the Goldreich-Ostrovsky bound is in terms of the *number of operations* that must be performed. With server computation, though the number of operations is still subject to the bound, most operations can be performed on the server-side without client intervention, making it possible to break the bound in terms of bandwidth between client and server. Since historically bandwidth has been the most important metric and the bottleneck for ORAM, breaking the bound in terms of bandwidth constitutes a significant advance.

However, it turns out that this is not easy. Indeed, two prior works [1,32] have made endeavors towards this direction using homomorphic encryption. Path-

PIR [32] leverages additively homomorphic encryption (AHE) to improve ORAM online bandwidth, but its overall bandwidth blowup is still poly-logarithmic. On the other hand, Apon et al. [1] showed that using a fully homomorphic encryption (FHE) scheme with *constant ciphertext expansion*, one can construct an ORAM scheme with constant bandwidth blowup. The main idea is that, instead of having the client move data around on the server "manually" by reading and writing to the server, the client can instruct the server to perform ORAM request and eviction operations under an FHE scheme without revealing any data and its movement. While this is a very promising direction, it suffers from the following drawbacks:

- First, ORAM keeps access patterns private by continuously shuffling memory as data is accessed. This means the ORAM circuit depth that has to be evaluated under FHE depends on the number of ORAM accesses made and can grow unbounded (which we say to mean any polynomial amount in N). Therefore, Apon et al. [1] needs FHE bootstrapping, which not only requires circular security but also incurs a large performance penalty in practice.[1]
- Second, with the server performing homomorphic operations on encrypted data, achieving malicious security is difficult. Consequently, most existing works either only guarantee semi-honest security [32,52], or leveraged powerful tools such as SNARKs to ensure malicious security [1]. However, SNARKs not only require non-standard assumptions [18], but also incur prohibitive cost in practice.

1.3 Our Contributions

With the above observation, the goal of this work is to construct constant bandwidth blowup ORAM schemes from *standard assumptions* that have *practical efficiency* and *verifiability in the malicious setting*. Specifically, we give proofs by construction for the following theorems. Let B be the block size in bits and N the number of blocks in the ORAM.

Theorem 1 (Semi-honest security construction). *Under the Decisional Composite Residuosity assumption (DCR) or Learning With Errors (LWE) assumption, there exists an ORAM scheme with semi-honest security, $O(B)$ bandwidth, $O(BN)$ server storage and $O(B)$ client storage. To achieve negligible in N probability of ORAM failure and success from best known attacks, our schemes require poly-logarithmic in N block size and server computation.*

We use negligible in N security following prior ORAM work but also give asymptotics needed for exact exponential security in Section 6.

Looking at the big picture, our DCR-based scheme is the first demonstration of a constant bandwidth blowup ORAM using any additively homomorphic

[1] While bootstrapping performance has been made asymptotically efficient by recent works [17], the cost in practice is still substantial, on the order of tens of seconds to minutes (amortized), whereas other homomorphic operations are on the order of milliseconds to seconds [22].

encryption scheme (AHE), as opposed to FHE. Our LWE-based scheme (detailed in the online version [9]) is the first time ORAM has been combined with SWHE/FHE in a way that does not require Gentry's bootstrapping procedure.

Our next goal is to extend our semi-honest constructions to the malicious setting. In Section 5, we will introduce the concept of "abstract server-computation ORAM" which both of our constructions satisfy. Then, we can achieve malicious security due to the following theorem:

Theorem 2 (Malicious security construction). *With the additional assumption of collision-resistant hash functions, any "abstract server-computation ORAM" scheme with semi-honest security can be compiled into a "verified server-computation ORAM" scheme which has malicious security.*

We stress that these are the *only* required assumptions. We do not need the circular security common in FHE schemes and do not rely on SNARKs for malicious security. We defer formal definitions of server-computation ORAM and malicious security to Appendix A.

Main ideas. The key technical contributions enabling the above results are:

- (Section 3) An ORAM that, when combined with server computation, has *shallow circuit depth*, i.e., $O(\log N)$ over the entire history of all ORAM accesses. This is a necessity for our constructions based on AHE or SWHE, and removes the need for FHE (Gentry's bootstrapping operations). We view this technique as an important step towards practical constant bandwidth blowup ORAM schemes.
- (Section 5) A novel technique that combines a cut and choose-like idea with an error-correcting code to amplify soundness.

Table 1 summarizes our contributions and compares our schemes with some of the state-of-the-art ORAM constructions.

Practical efficiency. To show how our results translate to practice, Section 6.4 compares our semi-honest AHE-based construction against Path PIR [32] and Circuit ORAM [47]—the best prior schemes with and without server computation that match our scheme in client/server storage. The top order bit is that as block size increases, our construction's bandwidth approaches $2B$. When all three schemes use an 8 MB block size (a proxy for modern image file size), Onion ORAM improves over Circuit ORAM and Path-PIR's bandwidth (in bits) by **35**× and **22**×, respectively. For larger block sizes, our improvement increases. We note that in many cases, block size is an application constraint: for applications asking for a large block size (e.g., image sharing), all ORAM schemes will use that block size.

1.4 Related Work

Recent non-server-computation ORAMs are approaching the Goldreich-Ostrovsky lower bound under $O(1)$ blocks of client storage. Goodrich et al. [21] and

Table 1: **Our contribution.** N is the number of blocks. The optimal block size is the data block size needed to achieve the stated bandwidth, and is measured in bits. All schemes have $O(B)$ client storage and $O(BN)$ server storage (both asymptotically optimal) and negligible failure probability in N. Computation measures the number of two-input plaintext gates evaluated per ORAM access. "M" stands for malicious security, and "SH" stands for semi-honest. We set parameters for AHE/SWHE (the Damgård-Jurik and Ring-LWE cryptosystems [3, 6], respectively) to get super-poly in N defense to best known attacks [26, 27]. For derivation of parameters for the SWHE schemes, see the extended version [9].

Scheme	Optimal Block size B	Bandwidth	Server Computation	Client Computation	Security
Circuit ORAM [47]	$\Omega(\log^2 N)$	$\omega(B \log N)$	N/A	N/A	M
Path-PIR [32]	$\omega(\log^5 N)$	$O(B \log N)$	$\widetilde{\omega}(B \log^5 N)$	$\widetilde{O}(B \log^4 N)$	SH
AHE Onion ORAM	$\widetilde{\Omega}(\log^5 N)$	$O(B)$	$\widetilde{\omega}(B \log^4 N)$	$\widetilde{O}(B \log^4 N)$	SH
	$\widetilde{\omega}(\log^6 N)$	$O(B)$	$\widetilde{\omega}(B \log^4 N)$	$\widetilde{O}(B \log^4 N)$	M
SWHE Onion ORAM	$\widetilde{\omega}(\log^2 N)$	$O(B)$	$\widetilde{\omega}(B \log^2 N)$	$\widetilde{\omega}(B)$	SH
	$\widetilde{\omega}(\log^4 N)$	$O(B)$	$\widetilde{\omega}(B \log^2 N)$	$\widetilde{\omega}(B + \log^2 N)$	M

Kushilevitz et al. [25] demonstrated $O(\log^2 N)$ and $O(\log^2 N / \log \log N)$ bandwidth blowup schemes, respectively. Recently, Wang et al. constructed Circuit ORAM [47], which achieves $\omega(\log N)$ bandwidth blowup.

Many state-of-the-art ORAM schemes or implementations make use of server computation. For example, the SSS construction [42, 43], Burst ORAM [8] and Ring ORAM [39] assumed the server is able to perform matrix multiplication or XOR operations. Path-PIR [32] and subsequent work [7, 52] increased the allowed computation to additively homomorphic encryption. Apon et al. [1] and Gentry et al. [13, 14] further augmented ORAM with Fully Homomorphic Encryption (FHE). Williams and Sion rely on server computation to achieve a single online roundtrip [49]. We remark that the techniques of Gentry et al. [13] and Wang et al. [46], for improving data structure performance on top of ORAM, can be combined with our techniques.

Recent works on Garbled RAM [15, 30] can also be seen as generalizing the notion of server-computation ORAM. However, existing Garbled RAM constructions incur $\mathsf{poly}(\lambda) \cdot \mathsf{polylog}(N)$ client work and bandwidth blowup, and therefore Garbled RAM does not give a server-computation RAM with constant bandwidth blowup. Reusable Garbled RAM [16] achieves constant client work and bandwidth blowup, but known reusable garbled RAM constructions rely on non-standard assumptions (indistinguishability obfuscation, or more) and are prohibitive in practice.

The mechanics of running our shallow depth ORAM over a homomorphic encryption scheme are similar to those used to evaluate encrypted branching programs [23]. (One may think of our contribution as formulating ORAM as a shallow enough circuit so that the techniques of [23] apply.)

2 Overview of Techniques

In our schemes, the client "guides" the server to perform ORAM accesses and evictions homomorphically by sending the server some "helper values". With these helper values, the server's main job will be to run a sub-routine called the "*homomorphic select*" operation (select operation for short), which can be implemented using either AHE or SWHE – resulting in two different constructions. We can achieve constant bandwidth blowup because helper value size is independent of data block size: when the block size sufficiently large, sending helper values does not affect the asymptotic bandwidth blowup. We now explain these ideas along with pitfalls and solutions in more detail. For the rest of the section, we focus on the AHE-based scheme but note that the story with SWHE is very similar.

Building block: homomorphic select operation. The select operation, which resembles techniques from private information retrieval (PIR) [27], takes as input m plaintext data blocks $\mathsf{pt}_1, \ldots, \mathsf{pt}_m$ and encrypted helper values which represent a user-chosen index i^*. The output is an encryption of block pt_{i^*}. Obviously, the helper values should not reveal i^*.

Our ORAM protocol will need select operations to be performed over the *outputs* of prior select operations. For this, we require a sequence of AHE schemes $\mathcal{E}_\ell$ with plaintext space $\mathbb{L}_\ell$ and ciphertext space $\mathbb{L}_{\ell+1}$ where $\mathbb{L}_{\ell+1}$ is again in the plaintext space of $\mathcal{E}_{\ell+1}$. Each scheme $\mathcal{E}_\ell$ is additively homomorphic meaning $\mathcal{E}_\ell(x) \oplus \mathcal{E}_\ell(y) = \mathcal{E}_\ell(x+y)$. We denote an ℓ-layer onion encryption of a message x by $\mathcal{E}^\ell(x) := \mathcal{E}_\ell(\mathcal{E}_{\ell-1}(\ldots \mathcal{E}_1(x)))$.

Suppose the inputs to a select operation are encrypted with ℓ layers of onion encryption, i.e., $\mathsf{ct}_i = \mathcal{E}^\ell(\mathsf{pt}_i)$. To select block i^*, the client sends an encrypted select vector (select vector for short), $\mathcal{E}_{\ell+1}(b_1), \ldots, \mathcal{E}_{\ell+1}(b_m)$ where $b_{i^*} = 1$ and $b_i = 0$ for all other $i \neq i^*$. Using this select vector, the server can homomorphically compute $\mathsf{ct}^* = \bigoplus_i \mathcal{E}_{\ell+1}(b_i) \cdot \mathsf{ct}_i = \mathcal{E}_{\ell+1}\left(\sum_i b_i \cdot \mathsf{ct}_i\right) = \mathcal{E}_{\ell+1}(\mathsf{ct}_{i^*}) = \mathcal{E}^{\ell+1}(\mathsf{pt}_{i^*})$. The result is the selected data block pt_{i^*}, with $\ell+1$ layers of onion encryption. Notice that the result has one more layer than the input.

All ORAM operations can be implemented using homomorphic select operations. In our schemes, for each ORAM operation, the client read/writes per-block metadata and creates a select vector(s) based on that metadata. The client then sends the encrypted select vector(s) to the server, who does the heavy work of performing actual computation over block contents.

Specifically, we will build on top of tree-based ORAMs [41, 45], a standard type of ORAM without server computation. Metadata for each block includes its logical address and the path it is mapped to. To request a data block, the client first reads the logic addresses of all blocks along the read path. After this step, the client knows which block to select and can run the homomorphic select protocol with the server. ORAM eviction operations require that the client sends encrypted select vectors to indicate how blocks should percolate down the ORAM tree. As explained above, each select operation adds an encryption layer to the selected block.

Achieving constant bandwidth blowup. To get constant bandwidth blowup, we must ensure that select vector bandwidth is smaller than the data block size. For this, we need several techniques. First, we will split each plaintext data block into C chunks $\mathsf{pt}_i = (\mathsf{pt}_i[1], \ldots, \mathsf{pt}_i[C])$, where each chunk is encrypted separately, i.e., $\mathsf{ct}_i = (\mathsf{ct}_i[1], \ldots, \mathsf{ct}_i[C])$ where $\mathsf{ct}_i[j]$ is an encryption of $\mathsf{pt}_i[j]$. Crucially, each select vector can be reused for all the C chunks. By increasing C, we can increase the data block size to decrease the relative bandwidth of select vectors.

Second, we require that each encryption layer adds a small *additive* ciphertext expansion (even a constant multiplicative expansion would be too large). Fortunately, we do have well established additively homomorphic encryption schemes that meet this requirement, such as the Damgård-Jurik cryptosystem [6]. Third, the "depth" of the homomorphic select operations has to be bounded and shallow. This requirement is the most technically challenging to satisfy, and we will now discuss it in more detail.

Bounding the select operation depth. We address this issue by constructing a new tree-based ORAM, which we call a "*bounded feedback ORAM*".[2] By "feedback", we refer to the situation where during an eviction some block a gets stuck in its current bucket b. When this happens, an eviction into b needs select operations that take both incoming blocks and block a as input, resulting in an extra layer on bucket b (on top of the layers bucket b already has). The result is that buckets will accumulate layers (with AHE) or ciphertext noise (with SWHE) on each eviction, which grows unbounded over time.

Our bounded feedback ORAM breaks the feedback loop by guaranteeing that bucket b will be empty at public times, which allows upstream blocks to move into b without feedback from blocks already in b. It turns out that breaking this feedback is not trivial: in all existing tree-based ORAM schemes [39, 41, 45, 47], blocks can get stuck in buckets during evictions which means there is no guarantee on when buckets are empty.[3] We remark that cutting feedback is equivalent to our claim of shallow circuit depth in Section 1.3: Without cutting feedback, the depth of the ORAM circuit keeps growing with the number of ORAM accesses.

Techniques for malicious security. We are also interested in achieving malicious security, i.e., enforcing honest behaviors of the server, while avoiding SNARKs. Our idea is to rely on probabilistic checking, and to leverage an error-correcting code to amplify the probability of detection. As mentioned before, each block is divided into C chunks. We will have the client randomly sample security parameter $\lambda \ll C$ chunks per block (the same random choice for all blocks), referred to as *verification chunks*, and use standard memory checking to ensure their authenticity and freshness. On each step, the server will perform homomorphic select operations on all C chunks in a block, and the client will

[2] Previous versions of this report used the term *"steady progress"* which has been cited in subsequent works, but we feel bounded feedback is more accurate.

[3] We remark that some hierarchical ORAM schemes (e.g., [20]) also have bounded feedback, but achieve worse results in different respects relative our construction (e.g., worse server storage, deeper select circuits), when combined with server computation.

Notation	Meaning
N	Number of real data blocks in ORAM
B	Data block size in bits
C	The number of chunks in each data block
B_C	Chunk size in bits ($B = C \cdot B_C$)
L	Depth of the ORAM tree
Z	Maximum number of real blocks per bucket
A	Eviction frequency (larger means less frequent)
$\mathcal{P}(l)$	The path from the root to leaf l
$\mathcal{P}(l, i)$	The i-th bucket (towards the root) on $\mathcal{P}(l)$
G	Eviction counter
S	The set of chunk indices corresponding to verification chunks

Table 2: ORAM parameters and notations.

perform the same homomorphic select operations on the λ verification chunks. In this way, whenever the server returns the client some encrypted block, the client can check whether the λ corresponding chunks match the verification chunks.

Unfortunately, the above scheme does not guarantee negligible failure of detection. For example, the server can simply tamper with a random chunk and hope that it's not one of the verification chunks. Clearly, the server succeeds with non-negligible probability. The fix is to leverage an error-correcting code to encode the original C chunks of each block into $C' = 2C$ chunks, and ensure that as long as $\frac{3}{4}C'$ chunks are correct, the block can be correctly decoded. Therefore, the server knows *a priori* that it will have to tamper with at least $\frac{1}{4}C'$ chunks to cause any damage at all, in which case it will get caught except with negligible probability.

3 Bounded Feedback ORAM

We now present the bounded feedback ORAM, a traditional ORAM scheme without server computation, to illustrate its important features. All notation used throughout the rest of the paper is summarized in Table 2.

3.1 Bounded Feedback ORAM Basics

We build on the tree-based ORAM framework of Shi et al. [41], 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. The leaves are numbered $0, 1, \ldots, 2^L - 1$ in the natural manner. Pseudo-code for our algorithm is given in Figure 1 and described below.

Note that many parts of our algorithm refer to *paths* down the tree where a path is a contiguous sequence of buckets from the root to a leaf. For a leaf bucket

l, we refer to the path to l as path l or $\mathcal{P}(l)$. $\mathcal{P}(l, k)$ denotes the bucket at level $k \in [0..L]$ on $\mathcal{P}(l)$. Specifically, $\mathcal{P}(l, 0)$ denotes the root, and $\mathcal{P}(l, L)$ denotes the leaf bucket on $\mathcal{P}(l)$.

Main invariant. Like all tree-based ORAMs, each block is associated with a random path and we say that each block can only live in a bucket along that path at any time. In a local position map, the client stores the path associated to each block.

Recursion. To avoid incurring a large amount of client storage, the position map should be recursively stored in other smaller ORAMs [41]. 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 (in terms of server storage or bandwidth blowup) are insignificant relative to the main ORAM [44]. Thus, for the remainder of the paper, we no longer consider the bandwidth cost of recursion.

Metadata. To enable all ORAM operations, each block of data in the ORAM tree is stored alongside its address and leaf label (the path the block is mapped to). This metadata is encrypted using a semantically secure encryption scheme.

ORAM Request. Requesting a block with address a (ReadPath in Figure 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 blocks on that path to find block a, invalidate block a, remap it to a new random path and add it to the root bucket. This involves decrypting the address metadata of every block on the path (Line 13) and setting one address to $\perp$ (Line 15). All addresses must be then re-encrypted to hide which block was invalidated.

ORAM Eviction. The goal of eviction is to percolate blocks towards the leaves to avoid bucket overflows and it is this procedure where we differ from existing tree-based ORAMs [13, 39, 41, 45, 47]. We now describe our eviction procedure in detail.

3.2 New Triplet Eviction Procedure

We combine techniques from [41], [13] and [39] to design a novel eviction procedure (Evict in Figure 1) that enables us to break select operation feedback.

Triplet eviction on a path. Similar to other Tree ORAMs, eviction is performed along a path. To perform an eviction: For every bucket $\mathcal{P}(l_e, k)$ (k from 0 to L, i.e., from root to leaf), we move blocks from $\mathcal{P}(l_e, k)$ to its two children. Specifically, each block in $\mathcal{P}(l_e, k)$ moves to either the left or right child bucket depending on which move keeps the block on the path to its leaf (this can be determined by comparing the block's leaf label to l_e). We call this process a bucket-triplet eviction.

In each of these bucket-triplet evictions, we call $\mathcal{P}(l_e, k)$ the *source bucket*, the child bucket also on $\mathcal{P}(l_e)$ the *destination bucket*, and the other child the *sibling bucket*. A crucial change that we make to the eviction procedure of the original

```
1: function Access(a, op, data′)
2:     l′ ← UniformRandom(0, 2^L − 1)
3:     l ← PositionMap[a]
4:     PositionMap[a] ← l′
5:     data ← ReadPath(l, a)
6:     if op = read then
7:         return data to client
8:     if op = write then
9:         data ← data′
10:    P(l, 0, cnt) ← (a, l′, data)
11:    Evict()

12: function ReadPath(l, a)
13:    Read all blocks on path P(l)
14:    Select and return the block with address a
15:    Invalidate the block with address a

16: function Evict( )
17:    Persistent variables cnt and G, initialized to 0
18:    cnt ← cnt + 1 mod A
19:    if cnt ≟ 0 then
20:        l_e ← bitreverse(G)
21:        EvictAlongPath(l_e)
22:        G ← G + 1 mod 2^L

23: function EvictAlongPath(l_e)
24:    for k ← 0 to L − 1 do
25:        Read all blocks in P(l_e, k) and its two children
26:        Move all blocks in P(l_e, k) to its two children
27:                                ▷ P(l_e, k) is empty at this point (Observation 1)
```

Fig. 1: Bounded Feedback ORAM (no server computation). Note that our construction differs from the original tree ORAM [41] only in the Evict procedure. We split Evict into EvictAlongPath to simplify the presentation later.

.

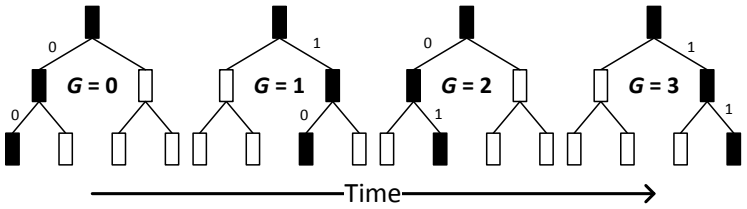

Fig. 2: The reverse lexicographical eviction order. **Black** buckets indicate those on each eviction path and G is the eviction count from Figure 1. As indicated in Figure 1, the eviction paths corresponding to $G = 4$ and $G = 0$ are equal: the exact eviction sequence shown above cycles forever. We mark the eviction path edges as 0/1 (goto left child = 0, right child = 1) to illustrate that the eviction path equals G in reverse binary representation.

binary-tree ORAM [41] is that we move *all* the blocks in the source bucket to its two children.

Eviction frequency and order. For every A (a parameter proposed in [39], which we will set later) ORAM requests, we select the next path to evict based on the reverse lexicographical order of paths (proposed in [13] and illustrated in Figure 2). The reverse lexicographical order eviction most evenly and *deterministically* spreads out the eviction on all paths in the tree. Specifically, a bucket at level k will get evicted *exactly* every $A \cdot 2^k$ ORAM requests.

Setting parameters for bounded feedback. As mentioned, we require that during a bucket-triplet eviction, *all* blocks in the source bucket move to the two child buckets. The last step to achieve bounded feedback is to show that child buckets will have enough room to receive the incoming blocks, i.e., no child bucket should ever overflow except with negligible probability. (If any bucket overflows, we have experienced ORAM failure.) We guarantee this property by setting the bucket size Z and the eviction frequency A properly. According to the following lemma, if we simply set $Z = A = \Theta(\lambda)$, the probability that a bucket overflows is $2^{-\Theta(\lambda)}$, exponentially small.

Lemma 1 (No bucket overflows). *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 $e^{-\frac{(2Z-A)^2}{6A}}$.*

The proof of Lemma 1 relies on a careful analysis of the stochastic process stipulated by the reverse lexicographic ordering of eviction, and boils down to a Chernoff bound. We defer the full proof to Appendix B.1. Now, Lemma 1 with $Z = A = \Theta(\lambda)$ immediately implies the following key observation.

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

Furthermore, straightforwardly from the definition of reverse lexicographical order, we have,

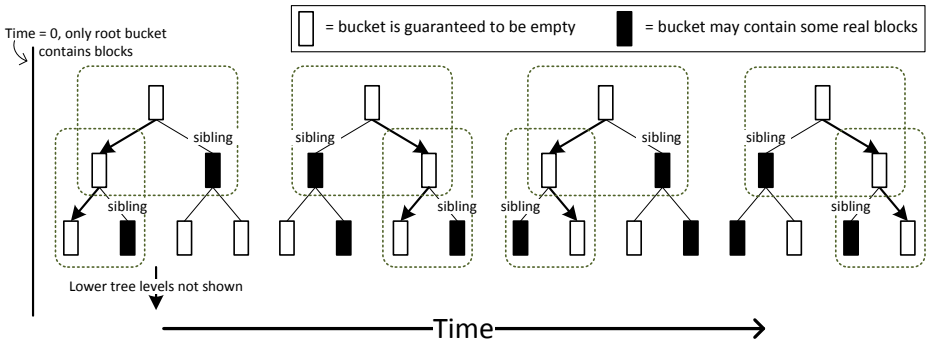

Fig. 3: ORAM tree state immediately after each of a sequence of four evictions. After an eviction, the buckets on the eviction path (excluding the leaves) are guaranteed to be empty. Further, at the start of each eviction, each sibling bucket for that eviction is guaranteed to be empty. **Notations:** Assume the ORAM tree has more levels (not shown for simplicity). The eviction path is marked with arrows. The dotted boxes indicate bucket triplets during each eviction.

Observation 2 *In reverse-lexicographic order eviction, each bucket rotates between the following roles in the following order: source, sibling, and destination.*

These observations together guarantee that buckets are empty at public and pre-determined times, as illustrated in Figure 3.

Towards bounded feedback. The above two observations are the keys to achieving bounded feedback. An empty source bucket b will be a sibling bucket the next time it is involved in a triplet eviction. So select operations that move blocks into b do not get feedback from b itself. Thus, the number of encryption layers (with AHE) or ciphertext noise (SWHE) becomes a function of previous levels in the tree only, which we can tightly bound later in Lemma 2 in Section 4.3.

Constant server storage blowup. We note that under our parameter setting $N \leq A \cdot 2^{L-1}$ and $Z = A$, our bounded feedback ORAM's server storage is $O(2^{L+1} \cdot Z \cdot B) = O(BN)$, a constant blowup.

4 Semi-Honest Onion ORAM with an Additively Homomorphic Encryption

In this section, we describe how to leverage an AHE scheme with additive ciphertext expansion to transform our bounded feedback ORAM into our semi-honest secure Onion ORAM scheme. First, we detail the homomorphic select operation that we introduced in Section 2.

4.1 Additively Homomorphic Select Sub-protocol

Suppose the client wishes to select the i^*-th block from m blocks denoted $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$, each with $\ell_1, \ldots, \ell_m$ layers of encryption respectively. The sub-protocol works as follows:

1. Let $\ell := \max(\ell_1, \ldots, \ell_m)$. The client creates and sends to the server the following encrypted select vector $\langle \mathcal{E}_{\ell+1}(b_1), \mathcal{E}_{\ell+1}(b_2), \ldots \mathcal{E}_{\ell+1}(b_m) \rangle$, where $b_{i^*} = 1$ and $b_i = 0$ for $i \neq i^*$.
2. The server "lifts" each block to ℓ-layer ciphertexts, simply by continually re-encrypting a block until it has ℓ layers $\mathsf{ct}'_i[j] = \mathcal{E}_\ell(\mathcal{E}_{\ell-1}(\ldots \mathcal{E}_{\ell_i}(\mathsf{ct}_i[j])))$.
3. The server evaluates the homomorphic select operation on the lifted blocks: $\mathsf{ct}_{out}[j] := \bigoplus_i (\mathcal{E}_{\ell+1}(b_i) \otimes \mathsf{ct}'_i[j]) = \mathcal{E}_{\ell+1}(\mathsf{ct}'_{i^*})$. The outcome is the selected block ct_{i^*} with $\ell + 1$ layers of encryption.

As mentioned in Section 2, we divide each block into C chunks. Each chunk is encrypted separately. All C chunks share the same select vector—therefore, encrypting each element in the select vector only incurs the chunk size (instead of the block size).

We stress again that every time a homomorphic select operation is performed, the output block gains an extra layer of encryption, on top of $\ell = \max(\ell_1, \ldots, \ell_m)$ onion layers. This poses the challenge of bounding onion encryption layers, which we address in Section 4.3.

4.2 Detailed Protocol

We now describe the detailed protocol. Recall that each block is tagged with the following metadata: the block's logical address and the leaf it is mapped to, and that the size of the metadata is independent of the block size.

Initialization. The client runs a key generation routine for all layers of encryption, and gives all public keys to the server.

Read path. $\mathsf{ReadPath}(l, a)$ from Section 3.1 can be done with the following steps:

1. Client downloads and decrypts the addresses of all blocks on path l, locates the block of interest a, and creates a corresponding select vector $\boldsymbol{b} \in \{0,1\}^{Z(L+1)}$.
2. Client and server run the homomorphic select sub-protocol with client's input being encryptions of each element in $\boldsymbol{b}$ and server's input being all encrypted blocks on path l. The outcome of the sub-protocol—block a—is sent to the client.
3. Client re-encrypts and writes back the addresses of all blocks on path l, with block a now invalidated. This removes block a from the path without revealing its location. Then, the client re-encrypts block a (possibly modified) under 1 layer, and appends it to the root bucket.

Eviction. To perform $\mathsf{EvictAlongPath}(l_e)$, do the following for each level k from 0 to $L - 1$:

1. Client downloads all the metadata (addresses and leaf labels) of the bucket triplet. Based on the metadata, the client determines each block's location after the bucket-triplet eviction.
2. For each slot to be written in the two child buckets:
   - Client creates a corresponding select vector $\boldsymbol{b} \in \{0,1\}^{2Z}$.
   - Client and server run the homomorphic select sub-protocol with the client's input being encryptions of each element in $\boldsymbol{b}$, and the server's input being the child bucket (being written to) and its parent bucket. Note that if the child bucket is empty due to Observation 1 (which is public information to the server), it conceptually has zero encryption layers.
   - Server overwrites the slot with the outcome of the homomorphic select sub-protocol.

4.3 Bounding Layers

Given the above protocol, we bound layers with the following lemma:

Lemma 2. *Any block at level $k \in [0..L]$ has at most $2k+1$ encryption layers.*

The proof of Lemma 2 is deferred to Appendix B.2. The key intuition for the proof is that due to the reverse-lexicographic eviction order, each bucket will be written to exactly twice (i.e., be a destination or sibling bucket) before being emptied (as a source bucket). Also in Appendix B.2, we introduce a further optimization called the "copy-to-sibling" optimization, which yields a tighter bound: blocks at level $k \in [0..L]$ will have only $k+1$ layers.

Eviction post-processing—peel off layers in leaf. The proof only applies to non-leaf buckets: blocks can stay inside a leaf bucket for an unbounded amount of time. Therefore, we need the following post-processing step for leaf nodes. After $\mathsf{EvictAlongPath}(l_e)$, the client downloads all blocks from the leaf node, peels off the encryption layers, and writes them back to the leaves as layer-$\Theta(L)$ re-encrypted ciphertexts (meeting the same layer bound as other levels). Since the client performs an eviction every A ORAM requests, and each leaf bucket has size $Z = A$, this incurs only $O(1)$ amortized bandwidth blowup.

4.4 Remarks on Cryptosystem Requirements

Let L' be the layer bound (derived in Section 4.3). For efficiency (in bandwidth for the overall protocol) we require the output of an arbitrary select operation performed during an ORAM request (note that $\ell = L'$ in this case) to be a constant times larger than the block size B. Since $L' = \omega(1)$, this implies we need additive blowup per encryption layer, independent of L'. One cryptosystem that satisfies the above requirement, for appropriate parameters, is the Damgård-Jurik cryptosystem (Section 6.2). We use this scheme to derive final parameters for the AHE construction in Section 6.

5 Security Against Fully Malicious Server

So far, we have seen an ORAM scheme that achieves security against an *honest-but-curious* server who follows the protocol correctly. We now show how to extend this to get a scheme that is secure against a fully malicious server who can deviate arbitrarily from the protocol.

5.1 Abstract Server-Computation ORAM

We start by describing several abstract properties of the Onion ORAM scheme from the previous section. We will call any server-computation ORAM scheme satisfying these properties an *abstract server-computation ORAM*.

Data blocks and metadata. The server storage consists of two types of data: *data blocks* and *metadata*. The server performs computation on data blocks, but never on metadata. The client reads and writes the metadata directly, so the metadata can be encrypted under any semantically secure encryption scheme.

Operations on data blocks. Following the notations in Section 2, each plaintext data block is divided into C chunks, and each chunk is separately encrypted $\mathsf{ct}_i = (\mathsf{ct}_i[1], \ldots, \mathsf{ct}_i[C])$. The client operates on the data blocks either by: (1) directly reading/writing an encrypted data block, or (2) instructing the server to apply a function f to form a new data block ct_i, where $\mathsf{ct}_i[j]$ only depends on the j-th chunk of other data blocks, i.e., $\mathsf{ct}_i[j] = f(\mathsf{ct}_1[j], \ldots, \mathsf{ct}_m[j])$ for all $j \in [1..C]$.

It is easy to check that the two Onion ORAM schemes are instances of the above abstraction. The metadata consists of the encrypted addresses and leaf labels of each data block, as well as additional space needed to implement ORAM recursion. The data blocks are encrypted under either a layered AHE scheme or a SWHE scheme. Function f is a "homomorphic select operation", and is applied to each chunk.

5.2 Semi-Honest to Malicious Compiler

We now describe a generic compiler that takes any "abstract server-computation ORAM" that satisfies honest-but-curious security and compiles it into a "verified server-computation ORAM" which is secure in the fully malicious setting.

Verifying metadata. We can use standard "memory checking" [2] schemes based on Merkle trees [33] to ensure that the client always gets the correct metadata, or aborts if the malicious server ever sends an incorrect value. A generic use of Merkle tree would add an $O(\log N)$ multiplicative overhead to the process of accessing metadata [29], which is good enough for us. This $O(\log N)$ overhead can also be avoided by aligning the Merkle tree with the ORAM tree [38], or using generic authenticated data structures [34]. In any case, verifying metadata is basically free in Onion ORAM.

Challenge of verifying data blocks. Unfortunately, we cannot rely on standard memory checking to protect the encrypted data blocks when the client doesn't read/write them directly but rather instructs the server to compute on them. The problem is that a malicious server that learns some information about the client's access pattern based on *whether the client aborts or not*.

Consider Onion ORAM for example. The malicious server wants to learn if, during the homomorphic select operation of a ORAM request, the location being selected is i. The server can perform the operation correctly except that it would replace the ciphertext at position i with some incorrect value. In this case, if the location being selected was indeed i then the client will abort since the data it receives will be incorrect, but otherwise the client will accept. This violates ORAM's privacy requirement.

A more general way to see the problem is to notice that the client's abort decision above depends on the decrypted value, which depends on the secret key of the homomorphic encryption scheme. Therefore, we can no longer rely on the semantic security of the encryption scheme if the abort decision is revealed to the server. To fix this problem, we need to ensure that the client's abort decision only depends on ciphertext and not on the plaintext data.

Verifying data blocks. For our solution, the client selects a random subset S consisting of λ chunk positions. This set S is kept secret from the server. The subset of chunks in positions $\{j : j \in S\}$ of every encrypted data block are treated as additional metadata, which we call the "verification chunks". Verification chunks are encrypted and memory checked in the same way as the other metadata. Whenever the client instructs the server to update an encrypted data block, the client performs the same operation himself on the verification chunks. Then, when the client reads an encrypted data block from the server, he can check the chunks in S against the ciphertexts of verification chunks. This check ensures that the server cannot modify too many chunks without getting caught. To ensure that this check is sufficient, we apply an error-correcting code which guarantees that the server has to modify a large fraction of chunks to affect the plaintext. In more detail:

- Every plaintext data block $\mathsf{pt} = (\mathsf{pt}[1], \ldots, \mathsf{pt}[C])$ is first encoded via an error-correcting code into a codeword block $\mathsf{pt\_ecc} = \mathsf{ECC}(\mathsf{pt}) = (\mathsf{pt\_ecc}[1], \ldots, \mathsf{pt\_ecc}[C'])$. The error-correcting code ECC has a rate $C/C' = \alpha < 1$ and can efficiently recover the plaintext block if at most a δ-fraction of the codeword chunks are erroneous. For concreteness, we can use a Reed-Solomon code, and set $\alpha = \frac{1}{2}, \delta = (1-\alpha)/2 = \frac{1}{4}$. The client then uses the "abstract server-computation ORAM" over the codeword blocks $\mathsf{pt\_ecc}$ (instead of pt).
- During initialization, the client selects a secret random set $S = \{s_1, \ldots, s_\lambda\} \subseteq [C']$. Each ciphertext data block ct_i has verification chunks $\mathsf{verCh}_i = (\mathsf{verCh}_i[1], \ldots, \mathsf{verCh}_i[\lambda])$. We ensure the invariant that, during an honest execution, $\mathsf{verCh}_i[j] = \mathsf{ct}_i[s_j]$ for $j \in [1..\lambda]$.
- The client uses a memory checking scheme to ensure the authenticity and freshness of the metadata including the verification chunks. If the client

detects a violation in metadata at any point, the client aborts (we call this abort_0).
- Whenever the client directly updates or instructs the server to apply the aforementioned function f on an encrypted data block ct_i, it also updates or applies the same function f on the corresponding verification chunks $\mathsf{verCh}_i[j]$ for $j \in [1..\lambda]$, which possibly involves reading other verification chunks that are input to f.
- When the client reads an encrypted data block ct_i, it also reads verCh_i and checks that $\mathsf{verCh}_i[j] = \mathsf{ct}_i[s_j]$ for each $j \in [1..\lambda]$ and aborts if this is not the case (we call this abort_1). Otherwise the client decrypts ct_i to get $\mathsf{pt\_ecc}_i$ and performs error-correction to recover pt_i. If the error-correction fails, the client aborts (we call this abort_2).

If the client ever aborts during any operation with $\mathsf{abort}_0, \mathsf{abort}_1$ or abort_2, it refuses to perform any future operations. This completes the compiler which gives us Theorem 2.

Security Intuition. Notice that in the above scheme, the decision whether abort_1 occurs does not depend on any secret state of the abstract server-computation ORAM scheme, and therefore can be revealed to the server without sacrificing privacy. We will argue that, if abort_1 does not occur, then the client retrieves the correct data (so abort_2 will not occur) with overwhelming probability. Intuitively, the only way that a malicious server can cause the client to either retrieve the incorrect data or trigger abort_2 without triggering abort_1 is to modify at least a δ (by default, $\delta = 1/4$) fraction of the chunks in an encrypted data block, but avoid modifying any of the λ chunks corresponding to the secret set S. This happens with probability at most $(1-\delta)^\lambda$ over the random choice of S, which is negligible. The complete proof is given in Appendix B.3.

6 Optimizations and Analysis

In this section we present two optimizations, an asymptotic analysis and a concrete (with constants) analysis for our AHE-based protocol.

6.1 Optimizations

Hierarchical Select Operation and Sorting Networks. For simplicity, we have discussed select operations as inner products between the data vector and the coefficient vector. As an optimization, we may use the Lipmaa construction [27] to implement select hierarchically as a tree of d-to-1 select operations for a constant d (say $d = 2$). In that case, for a given 1 out of Z selection, $\boldsymbol{b}^{\mathsf{hier}} \in \{0,1\}^{\log Z}$. Eviction along a path requires $O(\log N)$ bucket-triplet operations, each of which is a Z-to-Z permutation. To implement an arbitrary Z-to-Z permutation, we can use the Beneš sorting network, which consists of a total of $O(Z \log Z)$ 2-to-1 select operations per triplet.

At the same time, both the hierarchical select and the Beneš network add $\Theta(\log Z)$ layers to the output as opposed to a single layer. Clearly, this makes the layer bound from Lemma 2 increase to $\Theta(\log Z \log N)$. But we can set related parameters larger to compensate.

Permuted Buckets. Observe that on a request operation, the client and the server need to run a homomorphic select protocol among $O(\lambda \log N)$ blocks. We can reduce this number to $O(\lambda)$ using the permuted bucket technique from Ring ORAM [39] (similar ideas were used in hierarchical ORAMs [20]). Instead of reading all slots along the tree path during each read, we can randomly permute blocks in each bucket and only read/remove a block at a random looking slot (out of $Z = \Theta(\lambda)$ slots) per bucket. Each random-looking location will either contain the block of interest or a dummy block. We must ensure that no bucket runs out of dummies before the next eviction refills that bucket's dummies. Given our reverse-lexicographic eviction order, a simple Chernoff bound shows that adding $\Theta(A) = \Theta(\lambda)$ dummies, which increases bucket size by a constant factor, is sufficient to ensure that dummies do not run out except with probability $2^{-\Theta(\lambda)}$. We do not permute the root bucket since it will require additional techniques (and does not give much benefit). Therefore, a read path selects among $O(Z + \log N) = O(\lambda + \log N) = O(\lambda)$ blocks.

6.2 Damgård-Jurik Cryptosystem

We implement our AHE-based protocol over the Damgård-Jurik cryptosystem [6], a generalization of Paillier's cryptosystem [37]. Both schemes are 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 our onion encryptions, $\mathsf{sk}_i, \mathsf{pk}_i = \mathcal{G}_i()$ for $i \geq 0$.

We denote the integers mod n as $\mathbb{Z}_n$. The plaintext 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 $\mathbb{Z}_{n^{s_0+i}}$.

6.3 Asymptotic Analysis

We first perform the asymptotic analysis for exact exponential security. The results for negligible in N security in Table 1 is derived by setting $\lambda = \omega(\log N)$ and $\gamma = \Theta(\log^3 N)$ according to best known attacks [27].

Semi-Honest Case

Chunk size. The Damgård-Jurik cryptosystem encrypts a message of length γs_0 bits to a ciphertext of length $\gamma(s_0 + 1)$ bits, where γ is a parameter dependent on the security parameter λ, and s_0 is a user-chosen parameter. Using Beneš network, each ciphertext chunk accumulates $O(\log \lambda \log N)$ layers of encryption at the maximum. Suppose the plaintext chunk size is $B_c := \gamma s_0$, then at the maximum onion layer, the ciphertext size would be $\gamma(s_0 + O(\log \lambda \log N))$. Therefore, to ensure constant ciphertext expansion at all layers, it suffices to set $s_0 := \Omega(\log \lambda \log N)$ and chunk size $B_c := \Omega(\gamma \log \lambda \log N)$. This means ciphertext chunks and homomorphic select vectors are also $\Omega(\gamma \log \lambda \log N)$ bits.

Then we want our block size to be asymptotically larger than the select vectors at each step of our protocol (other metadata are much smaller).

Size of select vectors. Each read requires $O(\log \lambda)$ encrypted coefficients of $O(B_c)$ bits each. Eviction along a path requires $O(\log N)$ Beneš network (bucket-triplet operations), a total of $O(\lambda \log \lambda \log N)$ encrypted coefficients. Also recall that one eviction happens per $A = \Theta(\lambda)$ accesses. Therefore, the select vector size per ORAM access (amortized) is dominated by evictions, and is $\Theta(B_c \log \lambda \log N)$ bits.

Setting the block size. Clearly, if we set the block size to be $B := \Omega(B_c \log \lambda \log N)$, the cost of homomorphic select vectors could be asymptotically absorbed, thereby achieving constant bandwidth blowup. Since the chunk size $B_c = \Omega(\gamma \log \lambda \log N)$, we have $B = \Omega(\gamma \log^2 \lambda \log^2 N)$ bits.

Server computation The bottleneck of server computation is to homomorphically multiple a block with a encrypted select coefficient. In Damgård-Jurik, this is a modular exponentiation operation, which has $\widetilde{O}(\gamma^2)$ computational complexity for γ-bit ciphertexts. This means the *per-bit* computational overhead is $\widetilde{O}(\gamma)$. The server needs to perform this operation on $O(\lambda)$ blocks of size B, and therefore has a computational overhead of $\widetilde{O}(\gamma)O(B\lambda)$.

Client computation Client needs to decrypt $O(\log \lambda \log N)$ layers to get the plaintext block, and therefore has a computational overhead of $\widetilde{O}(\gamma)O(B \log \lambda \log N)$.

Malicious Case

Setting the block size. The main difference from semi-honest case is that on a read, the client must additionally download $\Theta(\lambda)$ verification chunks from each of the $\Theta(\lambda)$ blocks (assuming permuted buckets). Select vector size stays the same, and the error-correcting code increases block size by only a constant factor. Thus, the block size we need to achieve constant bandwidth over the entire protocol is $B = \Omega(B_c \lambda^2) = \Omega(\gamma \lambda^2 \log \lambda \log N)$.

Client computation. Another difference is that the client now needs to emulate the server's homomorphic select operation on the verification chunks. But a simple analysis will show that the bottleneck of client computation is still onion decryption, and therefore remains the same asymptotically.

6.4 Concrete Analysis (Semi-honest case only)

Figure 4 shows bandwidth as a function of block size for our optimized semi-honest construction, taking into account all constant factors (including the extra bandwidth cost to recursively look up the position map). Other scheme variants in this paper have the same general trend. We compare to Path PIR and Circuit ORAM, the most bandwidth-efficient constructions with/without server computation that match our server/client storage asymptotics.

Takeaway. The high order bit is that as block size increases, Onion ORAM's bandwidth approaches $2B$. Note that $2B$ is the inherent lower bound in bandwidth since every ORAM access must at least the block of interest from the server and send it back after possibly modifying it. Given an 8 MB block size, which is approximately the size of an image file, we improve in bandwidth over Circuit ORAM by **35**× and improve over Path PIR by **22**×. For very large block sizes, our improvement continues to increase but Circuit ORAM and Path PIR improve less dramatically because their asymptotic bandwidth blowup has a $\log N$ factor. Note that for sufficiently small block sizes, both Path PIR and Circuit ORAM beat our bandwidth because our select vector bandwidth dominates. Yet, this crossover point is around 128 KB, which is reasonable in many settings.

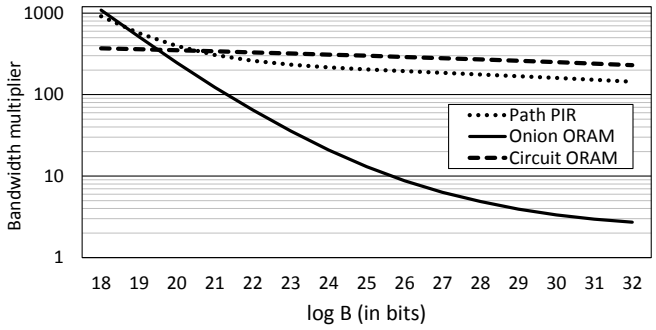

Fig. 4: Plots the bandwidth multiplier (i.e., the hidden constant for $O(B)$) for semi-honest Onion ORAM and two prior proposals. We fix the ORAM capacity to $NB = 2^{50}$ and give each scheme the same block size across different block sizes (hence as B increases, N decreases).

Constant factor optimization: Less frequent leaf post-processing. In the above evaluation, we apply an additional constant factor optimization. Since $Z = A = \Theta(\lambda)$, we must send and receive one additional data block (amortized) per ORAM request to post-process leaf buckets during evictions (Section 4.3). To save bandwidth, we can perform this post-processing on a particular leaf bucket every p evictions to that leaf (p is a free variable). The consequence is that the number of layers that accumulate on leaf buckets increases by p which makes

each ORAM read path more expensive by the corresponding amount. In practice, $p \geq 8$ yields the best bandwidth.

Parameterization details. For both schemes, we set acceptable ORAM failure probability to 2^{-80} which results in $Z = A \approx 300$ for Onion ORAM, $Z = 120$ for Path PIR [41] and a stash size (stored on the server) of 50 blocks for Circuit ORAM [47]. For Onion ORAM and Path PIR we set $\gamma = 2048$ bits. For Circuit ORAM, we use the reverse lexicographic eviction order as described in that work, which gives 2 evictions per access and $Z = 2$. For Path PIR, we set the eviction frequency $v = 2$ [41].

6.5 Other Optimizations and Remarks

De-Amortization. We remark that it is easy to de-amortize the above algorithm so that the worst-case bandwidth equals amortized bandwidth and overall bandwidth doesn't increase. First, it is trivial to de-amortize the leaf bucket post-processing (Section 4.3) over the A read path operations because $A = Z$ and post-processing doesn't change the underlying plaintext contents of that bucket. Second, the standard de-amortization trick of Williams et al. [50] can be applied directly to our EvictAlongPath operation. We remark that it is easy to de-amortize evictions over the next A read operations because moving blocks from buckets (possibly on the eviction path) to the root bucket does not impact our eviction algorithm.

Online Roundtrips. The standard recursion technique [44] uses a small block size for position map ORAMs (to save bandwidth) and requires $O(\log N)$ roundtrips. In Onion ORAM, the block in the main ORAM is large $B = \Omega(\lambda \log N)$. We can use Onion ORAM with the same large block size for position map ORAMs. This achieves a constant number of recursive levels if N is polynomial in λ, and therefore maintains the constant bandwidth blowup.

7 Conclusion and Open Problems

This paper proposes *Onion ORAM*, the first concrete ORAM scheme with optimal asymptotics in worst-case bandwidth blowup, server storage and client storage in the single-server setting. We have shown that FHE or SWHE are not necessary in constructing constant bandwidth ORAMs, which instead can be constructed using only an additively homomorphic scheme such as the Damgård-Jurik cryptosystem. Yet combining SWHE with Onion ORAM improves the computational efficiency of the scheme. We further extend Onion ORAM to be secure in the fully malicious setting using standard assumptions. Due to the known efficiency of SWHE schemes like BGV, we think of our work as an important step towards *practical* constant bandwidth blowup ORAM schemes.

We do note that while our block size is poly-logarithmic, the exponent is rather large (especially for our malicious construction). Subsequent to our proposal of Onion ORAM, Moataz et al. [35] combined our bounded feedback ORAM with

an optimized merge procedure for evictions which reduces server computation and block size for the semi-honest construction. We applaud this effort and argue that semi-honest constant bandwidth blowup ORAM is practical (or nearly practical). We leave tightening up poly-logarithmic factors for our malicious security construction as future work.

Beyond tightening parameters, an open problem is whether constant bandwidth blowup ORAMs can be constructed from non-homomorphic encryption schemes. The computational complexity of the Damgård-Jurik cryptosystem (which relies on modular exponentiation for homomorphic operations), or even more efficient SWHE schemes may be a bottleneck in practice. Can we construct constant bandwidth ORAM using simple computation such as XOR and any semantically secure encryption scheme with small ciphertext blowup? A partial result in this direction comes from Burst ORAM [8]: simple computation on ciphertexts (mod 2 XOR) enables a family of schemes (e.g., [39]) to achieve constant online bandwidth blowup on a *request*. Whether similar ideas can lead to constant bandwidth blowup on eviction is unclear.

Acknowledgements

We thank Vinod Vaikuntanathan for helpful discussion on this work.

References

1. D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In *PKC*. 2014.
2. M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *FOCS*, 1991.
3. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In *CRYPTO'11*, 2011.
4. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 2000.
5. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.
6. I. Damgard and M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *PKC*, 2001.
7. J. Dautrich and C. Ravishankar. Combining ORAM with PIR to minimize bandwidth costs. In *CODASPY*, 2015.
8. J. Dautrich, E. Stefanov, and E. Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *USENIX security*, 2014.
9. S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion oram: A constant bandwidth blowup oblivious ram. Cryptology ePrint Archive, Report 2015/005, 2015.
10. C. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based Oblivious RAM. In *ASPLOS*, 2015.
11. C. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, D. Serpanos, and S. Devadas. A low-latency, low-area hardware Oblivious RAM controller. In *FCCM*, 2015.

12. C. Fletcher, M. van Dijk, and S. Devadas. Secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.
13. 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 *PETS*, 2013.
14. C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345.
15. C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled RAM revisited. In *EUROCRYPT*, 2014.
16. C. Gentry, S. Halevi, M. Raykova, and D. Wichs. Outsourcing private RAM computation. In *FOCS*, 2014.
17. C. Gentry, S. Halevi, and N. P. Smart. Better bootstrapping in fully homomorphic encryption. In *PKC*, 2012.
18. C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, 2011.
19. O. Goldreich. Towards a theory of software protection and simulation on Oblivious RAMs. In *STOC*, 1987.
20. O. Goldreich and R. Ostrovsky. Software protection and simulation on Oblivious RAMs. In *Journal of the ACM*, 1996.
21. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless Oblivious RAM simulation. In *SODA*, 2012.
22. S. Halevi and V. Shoup. Bootstrapping for HElib. In *EUROCRYPT*, 2015.
23. Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In *Proceedings of TCC*, 2007.
24. M. Keller and P. Scholl. Efficient, Oblivious data structures for MPC. Cryptology ePrint Archive, Report 2014/137, 2014.
25. E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based Oblivious RAM and a new balancing scheme. In *SODA*, 2012.
26. R. Lindner and C. Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Topics in Cryptology - CT-RSA*, 2011.
27. H. Lipmaa. An Oblivious Transfer protocol with log-squared communication. In *ISC*, 2005.
28. C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient RAM-model secure computation. In *Oakland*, 2014.
29. J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *FAST*, 2013.
30. S. Lu and R. Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, 2013.
31. 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. In *CCS*, 2013.
32. T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS*, 2014.
33. R. C. Merkle. Protocols for public key cryptography. In *Oakland*, 1980.
34. A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *POPL*, 2014.
35. T. Moataz, T. Mayberry, and E.-O. Blass. Constant communication Oblivious RAM. Cryptology ePrint Archive, Report 2015/570, 2015.
36. R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.
37. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

38. L. Ren, C. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for Path Oblivious-RAM. In *HPEC*, 2013.
39. L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. V. Dijk, and S. Devadas. Constants count: Practical improvements to Oblivious RAM. In *USENIX security*, 2015.
40. 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 *ISCA*, 2013.
41. E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log n)^3)$ worst-case cost. In *ASIACRYPT*, 2011.
42. E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *S&P*, 2013.
43. E. Stefanov, E. Shi, and D. Song. Towards practical Oblivious RAM. In *NDSS*, 2012.
44. 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. Cryptology ePrint Archive, Report 2013/280.
45. 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 *CCS*, 2013.
46. X. Wang, K. Nayak, C. Liu, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. *IACR*, 2014.
47. 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.
48. X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *CCS*, 2014.
49. P. Williams and R. Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.
50. P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.
51. 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 *CCSW*, 2013.
52. 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.

A Definitions of Server-Computation ORAM

We directly adopt the definitions and notations used by Apon et al. [1] who are the first to define server-computation ORAM as a reactive two-party protocol between the client and the server, and define its security in the Universal Composability model [5]. We use the notation

$$((c\_out, c\_state), (s\_out, s\_state)) \leftarrow \mathsf{protocol}((c\_in,\ c\_state), (s\_in,\ s\_state))$$

to denote a (stateful) protocol between a client and server, where *c_in* and *c_out* are the client's input and output; *s_in* and *s_out* are the server's input and output; and *c_state* and *s_state* are the client and server's states before and after the protocol.

We now define the notion of a *server-computation ORAM*, where a client outsources the storage of data to a server, and performs subsequent read and write operations on the data.

Definition 1 (Server-computation ORAM). *A server-computation ORAM scheme consists of the following interactive protocols between a client and a server.*

- $((\perp, z), (\perp, Z)) \leftarrow \mathsf{Setup}(1^\lambda, (D, \perp), (\perp, \perp))$: An interactive protocol where the client's input is a memory array $D[1..n]$ where each memory *block* has bit-length β; and the server's input is $\perp$. At the end of the Setup protocol, the client has secret state z, and server's state is Z (which typically encodes the memory array D).
- $((\mathsf{data}, z'), (\perp, Z')) \leftarrow \mathsf{Access}((\mathsf{op}, z), (\perp, Z))$: To access data, the client starts in state z, with an input op where $\mathsf{op} := (\mathsf{read}, \mathit{ind})$ or $\mathsf{op} := (\mathsf{write}, \mathit{ind}, \mathsf{data})$; the server starts in state Z, and has no input. In a correct execution of the protocol, the client's output data is the current value of the memory D at location ind (for writes, the output is the old value of $D[\mathit{ind}]$ before the write takes place). The client and server also update their states to z' and Z' respectively. The client outputs $\mathsf{data} := \perp$ if the protocol execution aborted.

We say that a server-computation ORAM scheme is correct, if for any initial memory $D \in \{0,1\}^{\beta n}$, for any operation sequence $\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_m$ where $m = \mathrm{poly}(\lambda)$, an $\mathsf{op} := (\mathsf{read}, \mathit{ind})$ operation would always return the last value written to the logical location ind (except with negligible probability).

A.1 Security Definition

We adopt a standard simulation-based definition of secure computation [4], requiring that a real-world execution "simulate" an ideal-world (reactive) functionality $\mathcal{F}$.

Ideal world. We define an ideal functionality $\mathcal{F}$ that maintains an up-to-date version of the data D on behalf of the client, and answers the client's access queries.

- *Setup.* An environment $\mathcal{Z}$ gives an initial database D to the client. The client sends D to an ideal functionality $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ of the fact that the setup operation occurred as well as the size of the database $N = |D|$, but not of the data contents D. The ideal-world adversary $\mathcal{S}$ says ok or abort to $\mathcal{F}$. $\mathcal{F}$ then says ok or $\perp$ to the client accordingly.
- *Access.* In each time step, the environment $\mathcal{Z}$ specifies an operation $\mathsf{op} := (\mathsf{read}, \mathit{ind})$ or $\mathsf{op} := (\mathsf{write}, \mathit{ind}, \mathsf{data})$ as the client's input. The client sends op to $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ (without revealing to $\mathcal{S}$ the operation op). If $\mathcal{S}$ says ok to $\mathcal{F}$, $\mathcal{F}$ sends $D[\mathit{ind}]$ to the client, and updates $D[\mathit{ind}] := \mathsf{data}$ accordingly if this is a write operation. The client then forwards $D[\mathit{ind}]$ to the environment $\mathcal{Z}$. If $\mathcal{S}$ says abort to $\mathcal{F}$, $\mathcal{F}$ sends $\perp$ to the client.

Real world. In the real world, an environment $\mathcal{Z}$ gives an honest client a database D. The honest client runs the Setup protocol with the server $\mathcal{A}$. Then

at each time step, $\mathcal{Z}$ specifies an input $\mathsf{op} := (\mathsf{read}, ind)$ or $\mathsf{op} := (\mathsf{write}, ind, \mathsf{data})$ to the client. The client then runs the Access protocol with the server. The environment $\mathcal{Z}$ gets the view of the adversary $\mathcal{A}$ after every operation. The client outputs to the environment the data fetched or $\perp$ (indicating abort).

Definition 2 (Simulation-based security: privacy + verifiability). *We say that a protocol $\Pi_{\mathcal{F}}$ securely computes the ideal functionality $\mathcal{F}$ if for any probabilistic polynomial-time real-world adversary (i.e., server) $\mathcal{A}$, there exists an ideal-world adversary $\mathcal{S}$, such that for all non-uniform, polynomial-time environment $\mathcal{Z}$, there exists a negligible function* negl *such that*

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

At an intuitive level, our definition captures the privacy and verifiability requirements for an honest client (the client is never malicious in our setting), in the presence of a malicious server. The definition simultaneously captures *privacy* and *verifiability*. Privacy ensures that the server cannot observe the data contents or the access pattern. Verifiability ensures that the client is guaranteed to read the correct data from the server — if the server cheats, the client can detect it and abort the protocol.

B Proofs

B.1 Bounded Feedback ORAM: Bounding Overflows

We now give formal proofs to show that buckets do not overflow in bounded feedback ORAM except with negligible probability.

Proof. (of Lemma 1). 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 probability 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.

B.2 Onion ORAM: Bounding Layers of Encryption

To bound the layers of onion encryption, we consider the following abstraction. Suppose all buckets in the tree have a layer associated with it.

- The root bucket contains layer-1 ciphertexts.
- For a bucket known to be empty, we define bucket.layer := 0.
- Each bucket-triplet operation moves data from parent to child buckets. After the operation, child.layer := $\max\{\text{parent.layer}, \text{child.layer}\} + 1$.

Recall that we use the following terminology. The bucket being evicted from is called the *source*, its child bucket on the eviction path is called the *destination*, and its other child forking off the path is called the *sibling*.

Proof. (of Lemma 2). We prove by induction.
Base case. The lemma holds obviously for the root bucket.
Inductive step. Suppose that this holds for all levels $\ell < k$. We now show that this holds for level k. Let bucket denote a bucket at level k. We focus on this particular bucket, and examine bucket.layer after each bucket-triplet operation that involves bucket. It suffices to show that after each bucket-triplet operation involving bucket, it must be that bucket.layer $\leq 2k+1$. If a bucket-triplet operation involves bucket as a source, we call it a *source operation* (from the perspective of bucket). Similarly, if a bucket-triplet operation involves bucket as a destination or sibling, we call it a *destination operation* or a *sibling operation* respectively.

Based on Observation 1,

$$\text{bucket.layer} = 0 \qquad \text{(after each source operation)}$$

Since a sibling operation must be preceded by a source operation (if there is any preceding operation), bucket must be empty at the beginning of each sibling operation. By induction hypothesis, after each sibling operation, it must be that

$$\text{bucket.layer} \leq 2(k-1) + 1 + 1 = 2k \qquad \text{(after each sibling operation)}$$

[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.

Since a destination operation must be preceded by a sibling operation (if there is any preceding operation), from the above we know that at the beginning of a destination operation $\mathsf{bucket.layer}$ must be bounded by $2k$. Now, by induction hypothesis, it holds that

$$\mathsf{bucket.layer} \leq 2k + 1 \qquad \text{(after each destination operation)}$$

Finally, our post-processing on leaves where the client peels of the onion layers extends this lemma to all levels including leaves.

Copy-to-sibling optimization and a tighter layer bound An immediate implication of Observation 1 plus Observation 2 is that whenever a source evicts into a sibling, the sibling bucket is empty to start with because it was a source bucket in the last operation it was involved in. This motivates the following optimization: the server can simply copy blocks from the source bucket into the sibling. The client would read the metadata corresponding to blocks in the source bucket, invalidate blocks that do not belong to the sibling, before writing the (re-encrypted) metadata to the sibling.

This copy-to-sibling optimization avoids accumulating an extra onion layer upon writes into a sibling bucket. With this optimization and using a similar inductive proof, it is not hard to show a bucket at level k in the tree have at most $k + 1$ layers.

B.3 Malicious Security Proof

The Simulator. To simulate the setup protocol with some data of size N, the simulator chooses a dummy database D' of size N consisting of all 0s. It then follows the honest setup procedure on behalf of the client with database D'. To simulate each access operation, the simulator follows the honest protocol for reading a dummy index, say, $ind' = 0$, on behalf of the client.

During each operation, if the client protocol that's being executed by the simulator aborts then the simulator sends abort to $\mathcal{F}$ and stops responding to future commands on behalf of the client, else it gives ok to $\mathcal{F}$.

Sequence of Hybrids. We now follow a sequence of hybrid games to show that the real world and the simulation are indistinguishable:

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

Game 0. Let this be the real game $\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}$ with an adversarial server $\mathcal{A}$ and an environment $\mathcal{Z}$.

Game 1. In this game, the client also keeps a local copy of the correct metadata and data-blocks (in plaintext) that should be stored on the server. Whenever the client reads any (encrypted) metadata from the server during any operation, if the memory checking does not abort, then instead of decrypting the read metadata, the client simply uses the locally stored plaintext copy.

The only difference between Game 0 and Game 1 occurs if in Game 0 the memory checking does not abort, but the client retrieves the incorrect encrypted metadata, which happens with negligible probability by the security of memory checking. Therefore Game 0 and Game 1 are indistinguishable.

Game 2. In this game the client doesn't store the correct values of verCh_i with the encrypted metadata on the server, but instead replaces these with dummy values. The client still stores the correct values of verCh_i in the plaintext metadata stored locally, which it uses to do all of the actual computations.
Game 1 and Game 2 are indistinguishable by the CPA security of the symmetric-key encryption scheme used to encrypt metadata. We only need CPA security since, in Games 1 and 2, the client never decrypts any of the metadata ciphertexts.

Game 3. In this game, whenever the client reads an encrypted data block ct_i from the server, if abort_1 does not occur, instead of decrypting and decoding the encrypted data-block, the client simply uses local copy of the plaintext data-block.
The only difference between Game 2 and Game 3 occurs if at some point in time the client reads an encrypted data block ct_i from the server such that at least a δ fraction of the ciphertext chunks $\{\mathsf{ct}_i[j]\}$ in the block have been modified (so that decoding either fails with abort_2 or returns an incorrect value) but none of the chunks in locations $i \in S$ have been modified (so that abort_1 does not occur).
We claim that Game 2 and Game 3 are statistically indistinguishable, with statistical distance at most $q(1-\delta)^\lambda$, where q is the total number of operations performed by the client. To see this, note that in both games the set S is initially completely random and unknown to the adversarial server. In each operation i that the client reads an encrypted data-block, the server can choose some set $S'_i \subseteq [C']$ of positions in which the ciphertext chunks are modified, and if $S'_i \cap S = \emptyset$ the server learns this information about the set S and the game continues, else the client aborts and the game stops. The server never gets any other information about S throughout the game. The games 2 and 3 only diverge if at some point the adversarial server guesses a set S'_i of size $|S'_i| \geq \delta C'$ such that $S \cap S'_i = \emptyset$. We call this the "bad event". Notice that the sets S'_i can be thought of as being chosen non-adaptively at the beginning of the game prior to the adversary learning any knowledge about S (this is because we know in advance that the server will learn $S'_i \cap S = \emptyset$ for all i prior to the game ending). Therefore, the probability that the bad event happens in the j'th operation is

$$\Pr_S[S'_j \cap S = \emptyset] \leq \binom{(1-\delta)C'}{\lambda} \Big/ \binom{C'}{\lambda} \leq (1-\delta)^\lambda$$

where $S \subseteq [C']$ is a random subset of size $|S| = \lambda$. By the union bound, the probability that the bad event happens during some operation $j \in \{1, \ldots, q\}$ is at most $q(1-\delta)^\lambda$.

Game' 3. In this game, the client runs the setup procedure using the dummy database D' (as in the simulation) instead of the one given by the environment.

Furthermore, for each access operation, the client just runs a dummy operation consisting of a read with the index $ind' = 0$ instead of the operation chosen by the environment. (We also introduce an ideal functionality $\mathcal{F}$ in this world which is given the correct database D at setup and the correct access operations as chosen by the environment. Whenever the client doesn't abort, it forwards the outputs of $\mathcal{F}$ to the environment.)
Games 3 and Game' 3 are indistinguishable by the semi-honest Onion ORAM scheme. In particular, in both games whenever the client doesn't abort, the client reads the correct metadata and data blocks as when interacting with an honest server, and therefore follows the same protocols as when interacting with an honest server. Furthermore, the decision whether or not the client aborts in these games (with abort_0 or abort_1; there is no more abort_2) only depends on the secret set S and the internal state of the memory checking scheme, but is independent of any of the secret state or decryption keys of the underlying semi-honest Onion ORAM scheme. Therefore, the view of the adversarial server in these games can be simulated given the view of the honest server.

Game' 2,1,0. We define Game' i for $i = 0, 1, 2$ the same way as Game i except that the client uses the dummy database D' and the dummy operations (reads with index $idx' = 0$) instead of those specified by the environment.
The arguments that Game' $i + 1$ and Game' i are indistinguishable as the same as those for Game $i + 1$ and Game i. Finally, we notice that Game 0 is the ideal game $\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ with the simulator $\mathcal{S}$.

Putting everything together, we see that the real and ideal games $\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}$ and $\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ are indistinguishable as we wanted to show.