Onion ORAM: A Constant Bandwidth ORAM using Additively Homomorphic Encryption

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
⋈ University of Maryland – elaine@cs.umd.edu
∘ Northeastern University – wichs@ccs.neu.edu
∗ Lead authors

Abstract. We present Onion ORAM, a constant bandwidth Oblivious RAM (ORAM) that leverages poly-logarithmic server computation to circumvent the logarithmic ORAM lower bound. Our construction does not rely on Fully Homomorphic Encryption, but employs an efficient additive homomorphic encryption scheme such as the Damgård-Jurik cryptosystem. Homomorphic operations on the encrypted blocks introduce onion layers of encryption – hence the name Onion ORAM. We propose novel techniques to prove 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 ORAM (even for the semi-honest setting).

1 Introduction

Oblivious RAM (ORAM), initially proposed by Goldreich and Ostrovsky [16,17], 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 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 [7,25,31,34,35], secure processors [8,9,24,30–32,43] and secure multi-party computation [11,12,19,22,40].

1.1 Server Computation in ORAM

The ORAM model considered historically, starting with the work of Goldreich and Ostrovsky [16,17,28], assumed that the server acts as a simple storage device that allows the client to read and write data to it, but doesn't perform any computation otherwise. However, in many scenarios investigated by subsequent works [7, 34, 42] (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 [1,6,11,12,25,31,35,44], or reduce the number of online roundtrips [41]. Section 1.4 gives a more detailed narrative of the results achieved by these works. We remark that some prior works [1,25] 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 [11,12,25,31,35,41,44], without making a clear definitional distinction from standard ORAM. Apon et al. were the first to observe that such a distinction is warrented [1], not only for the extra rigor, but also because the definition renders the important Goldreich-Ostrovsky ORAM lower bound [17] 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 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 and server storage is the amount of storage needed at the server to store all data blocks.

In their seminal work [17], 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]. 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, many 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 for ORAM, breaking the bound in terms of bandwidth constitutes a significant advance.

With the above observation, the goal of this work is to construct a constant bandwidth overhead ORAM (overcoming the Goldreich-Ostrovsky lower-bound) that incurs only poly-logarithmic server computation. It turns out that achieving this is not easy. Indeed, two prior works [1, 25] have made endeavors towards this direction using homomorphic encryption. The construction in [25] uses additively homomorphic encryption (AHE) to improve ORAM online bandwidth, but still incurs poly-logarithmic overall bandwidth. On the other hand, the work of [1] showed that if a fully homomorphic encryption (FHE) scheme with *constant ciphertext expansion* existed, then one can hope to construct an ORAM scheme with constant bandwidth blowup. While this is a very promising direction, it suffers from the following drawbacks:

- Firstly, there is no *explicitly documented* construction of an FHE scheme with constant ciphertext expansion. Most FHE schemes encrypt the data bit-by-bit and therefore incur large multiplicative overhead which is polynomial in the security parameter. This overhead would immediately be reflected in the bandwidth blowup of the ORAM scheme. While it is conceivable that FHE with constant ciphertext expansion can be constructed by relying on ciphertext packing and bootstrapping techniques, it would require a delicate and non-trivial analysis, parametrization, and reliance on specific assumptions (rather than generic FHE techniques).
- Second, FHE schemes tend to incur a large performance penalty in practice. Therefore, we desire a construction that is implementable and efficient in practice, ideally without the use of FHE.
- Third, with the server performing homomorphic operations on data, achieving malicious security is difficult. Consequently, most existing works that leverage PIR or FHE techniques only guarantee semi-honest security [25, 44]. Apon et al. leveraged powerful tools such as SNARKs to ensure malicious security [1]; however, SNARKs not only require non-standard assumptions [15], but also incur prohibitive cost in practice.

1.3 Our Contributions

Based on Section 1.2, it is fair to conclude that to date, *no concrete instantiation of an ORAM scheme with constant bandwidth blowup has been documented, even in the semi-honest setting,* let alone one that considers practical efficiency and verifiability under standard assumptions.

In this paper, we achieve it all: we construct *Onion ORAM*, a server-computation ORAM with constant bandwidth blowup and poly-logarithmic server computation, with security against a *fully malicious* server. Our scheme also achieves *constant client storage* and *constant server storage blowup*. Moreover, we achieve these properties not only under standard assumptions, but also without the use of FHE [1, 25] or somewhat homomorphic encryption [12] (SWHE); we only rely on additively homomorphic encryption. We view our work as an important step towards realizing constant bandwidth overhead ORAM in practice.

Table 1: **Our contribution.** B is the ORAM data block size in bits. The reported asymptotics are achieved when B satisfies the optimal block size requirement. N is the number of blocks. Schemes in this table achieve $2^{-\lambda}$ failure probability ($\lambda = 80$ is a reasonable value). For Path-PIR and Onion ORAM, γ denotes the length of the modulus n of the Damgård-Jurik cryptosystem [5] ($\gamma = 2048$ is a reasonable value). Server computation measures the *amount of data* that is touched and computed upon by the server. "M" stands for malicious security, and "SH" stands for semi-honest.

Scheme	Optimal Block size	Bandwidth Cost	Client Storage	Server Storage	Server Computation	Security
Path ORAM [37]	$\Omega(\log^2 N)$	$O(B \log N)$	$O(B\lambda)$	$O(BN)$	N/A	M
Circuit ORAM [39]	$\Omega(\log^2 N)$	$O(B\lambda)$	$O(B)$	$O(BN)$	N/A	M
Path-PIR [25]	$O(\gamma\lambda \log N)$	$O(B\lambda)$	$O(B)$	$O(BN \log N)$	$O(B\lambda \log N)$	SH
Onion ORAM	$\Omega(\gamma \log^2 \lambda \log^2 N)$	$O(B)$	$O(B)$	$O(BN)$	$O(B\lambda \log N)$	SH
	$\Omega(\gamma\lambda \log \lambda \log^2 N)$	$O(B)$	$O(B)$	$O(BN)$	$O(B\lambda \log N)$	M

Table 1 summarizes our contributions and compares our schemes with some of the state-of-the-art ORAM constructions. We give a high-level overview of the construction and the techniques in Section 2. We defer formal definitions of ORAM with server computation and malicious security to Appendix A.

1.4 Related Work

In the standard ORAM setting with no server computation, Goldreich and Ostrovsky show that any ORAM scheme with constant client storage must incur at least $\Omega(\log N)$ blowup in terms of bandwidth and number of accesses [17]. Several recent constructions have been proposed in the standard ORAM setting that increasingly approached this lower bound. Kushilevitz et al. showed a construction with $O(\log^2 N / \log \log N)$ bandwidth blowup [20]. Stefanov et al. constructed Path ORAM, achieving $O(\log N)$ bandwidth blowup for $\Omega(\log^2 N)$-sized blocks, but requiring $O(\lambda)$ blocks[1] of client storage [37]. Recently, Wang et al. constructed Circuit ORAM [39], which achieves $O(\lambda)$ bandwidth blowup with $\Omega(\log^2 N)$ block size and $O(1)$ blocks of client storage.

Many state-of-the-art ORAM schemes or ORAM implementations make use of server computation. For example, the SSS construction [34, 35], Burst ORAM [7] and Ring ORAM [31] assumed the server is able to perform matrix multiplication or XOR operations. Path PIR [25] and subsequent work [6, 44] increased the allowed computation to additively homomorphic encryption such as Trostle-Parrish PIR [38] or Paillier's Cryptosystem [29]. Apon et al. [1], Path-PIR [25] and Gentry et al. [11, 12] further augmented ORAM with Fully Homomorphic Encryption (FHE). Williams and Sion rely on server computation to achieve a single online roundtrip [41].

Recent works on Garbled RAM [10, 13, 23] can also be seen as generalizing the notion of server-computation ORAM. However, existing Garbled RAM constructions incur $\mathsf{poly}(\lambda) \cdot \mathsf{poly} \log N$ client work and bandwidth blowup, and therefore Garbled RAM does not give a constant-bandwidth ORAM scheme with server computation. Reusable Garbled RAM [14] achieves constant client work and bandwidth blowup – however, known reusable garbled RAM constructions rely on non-standard assumptions (indistinguishability obfuscation, or more) and are prohibitive in practice.

[1] Throughout this paper, we set the desired failure probability to $2^{-\lambda}$. Many previous ORAM works just require the failure probability to be $\mathsf{negl}(N)$ in which case it suffices to set $\lambda = \omega(\log N)$. In such a parameterization, server computation is poly-logarithmic in the data size N.

2 Technical Roadmap and Highlights

2.1 Naive Attempt and Challenges

We start with the main ideas proposed in [1, 25]: 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 move data around under a homomorphic encryption scheme without revealing anything about the data and its movement.

To expand on the above, let us start with a tree-based ORAM scheme [33, 37] which is a standard ORAM without server computation. We assume that readers are familiar with the basics of tree-based ORAMs – for an overview see [33] or [37]. The idea is to encrypt all data on the server with FHE. During a read, the client wants to retrieve one block from a certain path identified by its leaf node, without revealing which block. Instead of having to download the entire ORAM tree path, the client can now instruct the server to perform a homomorphic computation which finds the correct block and sends back the FHE-encrypted result to the client. All eviction operations are homomorphically evaluated by the server, without the client's intervention. Finally, to avoid storing the position map on the client side, we can leverage the standard recursion technique [33], and parametrize it to preserve the same asymptotical bandwidth [36, 39].

Main problems. Unfortunately, as mentioned in Section 1.2, the above simple idea is deficient for two important reasons. First, we do not know of a concrete FHE instantiation with constant ciphertext expansion, and even if we had one it would most likely incur large computational overheads.

Second, this solution does not provide security against a malicious server. As an attack example, the server may guess that a read operation is requesting the same block as the previous read. To test this hypothesis, the server can simply send back the same homomorphically encrypted block as the previous read (perhaps re-randomized). Then, either the client gets the wrong data (if no authentication is performed), or the server learns if its hypothesis was correct based on whether the client rejects the data.

2.2 Overview of Our Construction

The starting point for our construction is to replace FHE with additively homomorphic encryption (AHE) to improve efficiency. This, however, introduces challenges. Without FHE, the server is no longer able to perform eviction operations entirely on its own. Instead, in our scheme, the client "guides" the server to perform evictions using only AHE, by sending the server some "helper values". At a high level, we can get constant bandwidth overhead because the size of these helper values is independent of the block size: by making the block size sufficiently large, sending helper values does not affect the asymptotic bandwidth overhead.

Building block: homomorphic select operation. Our Onion ORAM construction repeatedly makes use of an important building block called a "*homomorphic select*" operation, which can be thought of as the technique in private-information-retrieval (PIR) [21], and which is applied repeatedly.

We start with a sequence of additively homomorphic encryption 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 of the schemes $\mathcal{E}_\ell$ are additively homomorphic meaning $\mathcal{E}_\ell(x) \oplus \mathcal{E}_\ell(y) = \mathcal{E}_\ell(x+y)$. We define the following short-hand notation to denote an ℓ-layer onion encryption of a message x by $\mathcal{E}^\ell(x) := \mathcal{E}_\ell(\mathcal{E}_{\ell-1}(\ldots\mathcal{E}_1(x)))$.

The "homomorphic select operation" starts with the server storing m plaintext data blocks $\mathsf{pt}_1, \ldots, \mathsf{pt}_m$. Crucial to efficiency, each plaintext data block is broken up into C chunks $\mathsf{pt}_i = (\mathsf{pt}_i[1], \ldots, \mathsf{pt}_i[C])$, and each chunk is onion-encrypted separately, i.e., $\mathsf{ct}_i = (\mathsf{ct}_i[1], \ldots, \mathsf{ct}_i[C])$ where $\mathsf{ct}_i[j] = \mathcal{E}^\ell(\mathsf{pt}_i[j])$. To select the block pt_{i^*} at the secret index i^*, the client sends a small "encrypted select vector" $\mathcal{E}_{\ell+1}(b_1), \ldots, \mathcal{E}_{\ell+1}(b_m)$ where $b_{i^*} = 1$ and $b_i = 0$ for all other $i \neq i^*$. Using this encrypted select vector, the server can homomorphically compute $\mathsf{ct}^* = (\mathsf{ct}^*[1], \ldots, \mathsf{ct}^*[C])$ where $\mathsf{ct}^*[j] = \bigoplus_i \mathcal{E}_{\ell+1}(b_i) \cdot \mathsf{ct}_i[j] = \mathcal{E}_{\ell+1}(\sum_i b_i \cdot \mathsf{ct}_i[j]) = \mathcal{E}_{\ell+1}(\mathsf{ct}_{i^*}[j]) = \mathcal{E}^{\ell+1}(\mathsf{pt}_{i^*}[j])$ for all $j \in [1..C]$. The result is selected data block pt_{i^*}, with $\ell+1$ layers of onion encryption.

We make two key remarks that are crucial to the efficiency of our scheme (more details in the next two paragraphs). First, the encrypted select vector is reused for all the C chunks. Therefore, if we set C sufficiently large (implying large data blocks), the bandwidth taken by transmitting the encrypted select vector is overshadowed by that of reading a single block.

Second, we need to ensure that the number of layers on each block does not grow unbounded, and that each layer adds a small *additive* ciphertext expansion (even a constant multiplicative expansion would be too large). For the former, we propose novel ORAM eviction techniques to bound the number of layers (outlined below). The latter requirement can be accomplished by the Damgård and Jurik additively homomorphic cryptosystem [5].

Homomorphic select and block-size independent client operations. Our Onion ORAM construction is a new tree-based ORAM that expresses ORAM operations in terms of the aforementioned "homomorphic select" operation. At a high level, each Onion ORAM operation has the client read/write per-block metadata and create an encrypted 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.

To elaborate, each block has a metadata entry that stores its address. During a read operation, the client first reads the metadata associated with all blocks along the read path. After this step, the client knows which block to select and sends a homomorphically encrypted select vector to the server, who proceeds to evaluate the homomorphic select operation over the actual block contents. ORAM eviction operations are more sophisticated but require similar interactions. After downloading metadata, the client sends a small number of encrypted select vectors to the server that move blocks through the ORAM tree. We describe the detailed protocol in Section 4.3. Once again, we use the idea that a sufficiently large block size can absorb all the cost of sending/receiving metadata and encrypted select vectors.

Bounding the layers of encryption. To ensure the efficiency of our scheme, it is important to ensure that the layers of onion encryption are bounded. As described above, a block accumulates an extra layer of encryption every time a "homomorphic select" operation is performed on that block during eviction. Roughly speaking, the property we wish to guarantee is "*steady progress*" – a block should advance along the eviction path every $O(1)$ times it is operated on during eviction. It turns out that achieving this property is non-trivial. In particular, in all existing tree-based ORAM schemes [31, 33, 37, 39], a block can be "*stagnant*", i.e., be involved in many eviction operations without advancing along the tree path. To address this, we propose a new tree-based ORAM construction (without server computation) which we prove to guarantee the "steady progress" requirement.[2]

Techniques for malicious security. Our last goal is to achieve malicious security, i.e., enforce honest behaviors of the server, while avoiding SNARKs [1].

Our idea is to rely on probabilistic checking, and to leverage an error-correcting code to amplify the probability of detection. Specifically, observe that in our underlying Onion ORAM scheme, the server only operates on *block data*, but never *metadata*. The metadata is always updated by the client itself. Further, each block is divided into chunks, and the server always applies the "same" operation to all chunks of a block. Now imagine the following modification to the scheme:

The client randomly samples λ chunks per block (the same random choice for all blocks), referred to as *verification chunks* and treats these λ chunks as part of the metadata. The client will use standard memory checking to ensure the authenticity of all metadata, including verification chunks. Whenever the client asks the server to perform homomorphic select operations on its behalf, the client will perform the same homomorphic select operations on the verification chunks itself. In this way, whenever the server returns the client some encrypted block, the client can easily corroborate its correctness by comparing the λ corresponding chunks with the λ verification chunks.

However, the above simple probabilistic checking is insufficient to guarantee $1 - \mathsf{negl}(\lambda)$ probability of detection when the server cheats. Specifically, the server can randomly guess that the i-th chunk is not one of the λ verification chunks, and tamper with it. Clearly, the server's guess is right with non-negligible probability. We need an additional technique to make the above idea fully work: we leverage an error-correcting code to amplify the probability of detection. The error-correcting code encodes the original C chunks of each block into $C' = 2C$ chunks, and ensures that as long as $\frac{3}{4}C'$ chunks are correct, the block can be correctly decoded.

[2] We remark that hierarchical ORAM schemes [17, 18, 20] also meet the "steady progress" requirement but achieve worse results in different respects than our construction, when combined with server computation.

Therefore, the server knows apriori that it will have to tamper with at least a quarter of the C' chunks to cause any damage at all, in which case it will get caught except with negligible probability.

3 Pre-Onion ORAM

We now present the underlying ORAM scheme without the encryption onions to illustrate the important features. We refer to this as the Pre-Onion ORAM scheme and will construct the final Onion ORAM protocol directly on top of it. We remark that Pre-Onion ORAM is not the most competitive ORAM scheme under the standard ORAM definition (i.e., no server computation). One of our primary goals here to craft a tree-based ORAM scheme that ensures the "steady progress" property (see Section 2 for an intuitive motivation), such that we can later bound the number of onion encryption layers in the full Onion ORAM.

3.1 Pre-Onion ORAM Basics

We build on the tree-based ORAM framework of Shi et al. [33], 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 Algorithm 1 and described below.

Main invariant. Like all tree-based ORAMs, each block is associated with a random path from the root to a leaf. In a local position map, the client stores the position of each block, i.e., the path where the block resides.

Recursion. To avoid incurring a large amount of client storage, the position map should 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 [36]. Thus, for the remainder of the paper, we will assume that the (recursed) position map does not introduce extra asymptotical cost in terms of server storage or bandwidth.

ORAM Read. Reading a block with address a in Algorithm 1 (ReadPath) 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 (our construction does not need a *stash*).

ORAM Eviction. Tree-based ORAMs are mostly distinguished by their eviction algorithms. The goal of eviction is to percolate blocks towards the leaves to avoid bucket overflows. The eviction algorithms of existing tree-based ORAMs [31, 33, 37, 39] do not ensure the steady progress property. In Section 3.2, we propose a novel eviction algorithm (EvictAlongPath) that does achieve this property.

Metadata. We note that both reads and evictions in Algorithm 1 are implemented with the help of metadata. In the ORAM tree, each block is stored alongside its address and leaf label (the path the block is mapped to) – see Line 11. Reading/invalidating a block is done by comparing the address of every block on the path with the address of interest a (Line 19) and changing its address field to $\perp$ (Line 20), an address reserved for dummy blocks. This requires decrypting/re-encrypting the address field of every block on the path to hide the block being invalidated. Likewise, how blocks percolate down the tree during an eviction depend on each block's leaf label.

3.2 New Triplet Eviction Algorithm and Steady Progress

We combine techniques from [33], [11] and [31] to design a novel eviction algorithm (EvictAlongPath in Algorithm 1) that guarantees "steady progress", which will later enable us to tightly bound the layers of onion encryption in Section 4.

Algorithm 1 Pre-Onion ORAM (no server computation). The text in box highlights the our key ideas for ensuring steady progress.

```
1: function Access(a, op, data')
2:     Global/persistent variables: cnt and G, initialized to 0
3:     l' ← UniformRandom(0, 2^L − 1)
4:     l ← PositionMap[a]
5:     PositionMap[a] ← l'
6:     data ← ReadPath(l, a)
7:     if op = read then
8:         return data to client
9:     if op = write then
10:        data ← data'
11:    P(l, 0, cnt) ← (a, l', data)                    ▷ add this block to the root
12:    cnt ← cnt + 1 mod A
13:    if cnt ≟ 0 then
14:        l_g ← bitreverse(G mod 2^L)                 ▷ reverse lexicographical order
15:        EvictAlongPath(l_g)
16:        G ← G + 1
17: function ReadPath(l, a)
18:    Read all blocks on path P(l)
19:    Select and return the block with address a
20:    Invalidate the block with address a
           ▷ Involves re-encrypting metadata of blocks on P(l) to hide which block is invalidated
21: function EvictAlongPath(l_g)
22:    for k ← 0 to L − 1 do
23:        Read all the blocks in P(l_g, k) and its two children
24:        Move all blocks in P(l_g, k) to its two children
25:                                   ▷ P(l_g, k) is guaranteed to be empty at this point (Observation 1)
```

Triplet eviction on a path. In Pre-Onion ORAM, eviction is performed along a path. A path from the root to a leaf l_g is referred to as the path l_g. We use the following notation to denote buckets on a path: $\mathcal{P}(l_g, k)$ denotes the bucket at level $k \in [0..L]$ on the path l_g. Specifically, $\mathcal{P}(l_g, 0)$ denotes the root, and $\mathcal{P}(l_g, L)$ denotes the leaf bucket.

To perform an eviction: For every bucket $\mathcal{P}(l_g, k)$ (k from 0 to L, i.e., from root to leaf) on path l_g, we move blocks from $\mathcal{P}(l_g, k)$ to its two children. We call this process a bucket-triplet eviction. In each of these bucket-triplet evictions, we call $\mathcal{P}(l_g, k)$ the *source bucket*, the child bucket also on $\mathcal{P}(l_g)$ the *destination bucket*, and the other child the *sibling bucket*.

A crucial change that we make to the eviction procedure of the original binary-tree ORAM [33] is that we move *all* the blocks in the source bucket to its two children. This is important for ensuring the "steady progress" property as we show later.

Eviction frequency and order. For every A (a parameter proposed in [31], which we will set later) ORAM reads, we select the next path to evict based on the reverse lexicographical order of paths (proposed in [11]). The idea is that the reverse lexicographical order eviction most evenly and *deterministically* spreads out the eviction on all buckets in the tree. Specifically, a bucket at level k will get evicted *exactly* every $A \cdot 2^k$ ORAM reads.

Setting parameters. To get "steady progress," each bucket-triplet eviction needs to write *all* blocks in a parent bucket to its two children, i.e., no block should get stuck. Therefore, it is important to show that the child buckets will have enough room to receive the incoming blocks, i.e., no child bucket should ever overflow except with negligible probability. 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 $Z = A = \Theta(\lambda)$, the probability that a bucket overflows is $2^{-\Theta(\lambda)}$, exponentially small.

Theorem 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 Theorem 1 relies on a careful analysis of the stochastic process stipulated by the Pre-Onion ORAM's random path reads and reverse lexicographic ordering of eviction, and eventually boils down to proving Chernoff-like bounds. We defer the full proof to Appendix C.1. Now, Theorem 1 with $Z = A = \Theta(\lambda)$ immediately implies the following key observation.

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

Towards bounding layers. Observation 1 is the key to bounding layers in our construction. It leaves the source bucket empty so the number of layers on it can be reset to 0. As alluded to earlier, it also ensures steady progress: the blocks in a source bucket are emptied into the two children buckets, making progress towards their leaves. One can also see from the reverse lexicographic eviction order that as a block progresses towards the leaves, it will be involved in an eviction exponentially less often. The above properties combined will allow us to bound the layers of onion encryption in the final scheme.

Constant server storage blowup. We conclude this section by noting that under our parameter setting $Z = A = \Theta(\lambda)$ and $N \leq A \cdot 2^{L-1}$, (Pre-)Onion ORAM's server storage is $O(2^{L+1} \cdot Z) = O(N)$ blocks, a constant blowup.

4 Onion ORAM with Semi-Honest Security

In this section, we describe how to leverage an additively homomorphic encryption scheme with additive ciphertext expansion (e.g., the Damgård-Jurik cryptosystem [5]) to transform our Pre-Onion ORAM into our semi-honest secure Onion ORAM scheme, by leveraging the homomorphic select operation we introduced in Section 2.

4.1 Intuition and Overview

Observe that in our Pre-Onion ORAM,

- **A read can be expressed as a select operation.** Each read operation is a select operation over the read path. When the client downloads the metadata (the addresses of all blocks on the path), it can create the selection vector.
- **A bucket-triplet eviction can be expressed as select operations.** In each bucket-triplet eviction, a parent bucket is evicted to its two children. This requires writing to $Z = O(\lambda)$ slots in the child buckets. Each write can be expressed as a select operation over the parent bucket and the child bucket itself. Similarly, the client only needs to read metadata associated with the bucket triplet to determine the selection vector for each write.

These observations allow the client to perform all read and eviction operations using the *block-size independent* homomorphic selection sub-protocol from Section 2. Since the client only sends/receives metadata and homomorphic select vectors, we can achieve $O(1)$ bandwidth blowup when the block size is sufficiently large, i.e., asymptotically greater than or equal to the metadata and the homomorphic select vectors.

4.2 Homomorphic Select Sub-protocol

We define the following sub-protocol between the client and the server. Suppose the client wishes to perform a homomorphic select operation on m blocks denoted $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$, each with $\ell_1, \ldots, \ell_m$ layers of encryption respectively.

Let $\boldsymbol{b} := (b_1, \ldots, b_m)$ denote the selection vector where $b_{i^*} = 1$ for the selection index i^* and $b_i = 0$ for $i \neq i^*$. As mentioned in Section 2, we divide each block into C *chunks* of B_c bits each. Each chunk is encrypted separately using the additively homomorphic encryption scheme. All C chunks share the same homomorphic select vector $\boldsymbol{b}$ – therefore, each element in $\boldsymbol{b}$ incurs roughly the chunk size B_c (instead of the block size).

The sub-protocol works as follows:

1. Let $\ell := \max(\ell_1, \ldots, \ell_m)$.
2. The client first creates and sends to the server the following homomorphic select vector $\langle \mathcal{E}_{\ell+1}(b_1), \mathcal{E}_{\ell+1}(b_2), \ldots \mathcal{E}_{\ell+1}(b_m) \rangle$.
3. The server "lifts" each ciphertext block (containing C chunks) to ℓ-layer ciphertexts, simply by continually re-encrypting a block (i.e., all its chunks) until it has ℓ layers $\mathsf{ct}'_i = \mathcal{E}_\ell(\mathcal{E}_{\ell-1}(\ldots \mathcal{E}_{\ell_i}(\mathsf{ct}_i)))$
4. Then, the server evaluates the following homomorphic select operation on all chunks $j \in [1..C]$ (of the lifted blocks):
$$\mathsf{ct}_{out}[j] := \bigoplus_i (\mathcal{E}_{\ell+1}(b_i) \otimes \mathsf{ct}'_i[j]) \; = \mathcal{E}_{\ell+1}(\mathsf{ct}'_{i^*})$$
5. The outcome of this sub-protocol is selected the block ct_{i^*} where all chunks are $(\ell+1)$-layer ciphertexts.

We stress that every time a homomorphic select operation is performed on blocks, the resulting block gains an extra layer of encryption, on top of $\max(\ell_1, \ldots, \ell_m)$ onion layers. This poses the challenge of bounding onion encryption layers, which we address in Section 4.4.

4.3 Detailed Protocol

We now describe the detailed protocol. Recall that each block is tagged with the following metadata: *i)* the block's logical address; *ii)* leaf label. Dummy blocks are represented by the reserved address $\perp$. 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. $\mathsf{ReadPath}(l, a)$ can be done with the following steps:

1. Client looks up the position map to determine the path $l := \mathsf{PositionMap}[a]$ to read.
2. Client to server: l
3. Server to client: (encrypted) addresses of all blocks on path l.
4. Client computation: decrypts addresses, locates the block a of interest, and creates a corresponding selection vector $\boldsymbol{b} \in \{0,1\}^{Z(L+1)}$.
5. Client and server run the homomorphic selection sub-protocol with client's input being $\boldsymbol{b}$ and server's input being all encrypted blocks on the path l.
6. Server to client: outcome of the homomorphic select sub-protocol – block a.
7. Client writes back all (re-encrypted) addresses with block a now invalidated. This removes block a from the path.
8. Client to server: re-encryption of block a (possibly modified if the client was performing a write) under 1 layer. The server appends this new ciphertext to the root bucket.

Evict along path. To perform $\mathsf{EvictAlongPath}(l_g)$ on a path l_g, do the following for each level k from 0 to $L-1$,

1. Server to client: all the metadata (addresses and leaf labels) of the bucket triplet, i.e., $\mathcal{P}(l_g, k)$ and its two children.
2. Client: Based on the metadata obtained, determine the location of each block after the bucket-triplet eviction.
3. For each slot to be written in $\mathcal{P}(l_g, k)$'s child buckets:
   - Client creates a corresponding selection vector $\boldsymbol{b} \in \{0,1\}^{2Z}$.
   - Client and server run the homomorphic selection sub-protocol with the client's input being $\boldsymbol{b}$, and the server's input being the child bucket (being written to) and its parent bucket.
   - Server overwrites the slot with the outcome of the homomorphic select sub-protocol.

4.4 Bounding Layers

Based on the steady progress property of Pre-Onion ORAM, we can prove (in Theorem 2) that all blocks at a non-leaf level k in the tree will accumulate only $O(k)$ layers of encryption.

Theorem 2 (Bounding layers.). *In our Onion ORAM scheme, any block at level $k \in [0..L]$ has at most $2k+1$ layers of onion encryption.*

The proof of Thereom 2 is deferred to Appendix C.2. The key intuition for the proof is that due to the reverse-lexicographic eviction order, each bucket will be written to (i.e., be a destination or sibling bucket in an eviction triplet) exactly twice before being evicted itself (as a source bucket in an eviction). Our proof only applies to non-leaf buckets: blocks can stay inside a leaf bucket for an unbounded amount of time. Therefore, we need an additional post-processing step for leaf nodes.

Eviction post-processing: peel off layers in leaf. After $\mathsf{EvictAlongPath}(l_g)$, 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 a path eviction every $A = \Theta(\lambda)$ ORAM requests, and each leaf bucket has size $Z = \Theta(\lambda)$ as well, this incurs only $O(1)$ amortized bandwidth blowup. Since this operation doesn't change the underlying plaintexts for each block, it is easy to de-amortize this cost among the next A requests to get $O(1)$ worst-case bandwidth blowup.

Further optimizations. Also in Appendix C.2, we introduce a further optimization called the "copy-to-sibling" optimization, which yields a tighter bound: blocks at level $k \in [0..L]$ of the tree will only have $k+1$ layers.

4.5 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 give additional details for in Appendix B.

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. In Onion ORAM, each ciphertext chunk accumulates $\Theta(\log N)$ layers of encryption at the maximum (Section 4.4). Suppose the plaintext chunk size is $B_c := \gamma s_0$, then at the maximum onion layer, the ciphertext size would be $\gamma(s_0 + \Theta(\log N))$. Therefore, to ensure constant ciphertext expansion at all layers, it suffices to set $s_0 := \Omega(\log N)$ and chunk size $B_c := \Omega(\gamma \log N)$. This means ciphertext chunks and homomorphic select vectors are also $\Omega(\gamma \log N)$ bits.

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 selection vector on reads. Reads require a select operation over $\Theta(ZL) = \Theta(\lambda \log N)$ blocks (note that $Z = A = \Theta(\lambda)$). Thus, the homomorphic select vector on a read is $\Theta(B_c \lambda \log N)$ bits.

Size of selection vectors on evictions. Eviction along a path requires $\Theta(\log N)$ bucket-triplet operations, each of which contains $\Theta(Z)$ select operations each selecting among $\Theta(Z)$ blocks. Also recall that one eviction happens per A accesses. Therefore, the selection vector size (amortized over $A = \Theta(\lambda)$ reads) for eviction is $\Theta(B_c \lambda \log N)$ bits. 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.

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

Optimization: hierarchical select operations. For simplicity, we have discussed select operations as inner products. We may also use the Lipmaa construction [21] 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, $\boldsymbol{b}^{\mathsf{tree}} \in \{0,1\}^{\log Z}$. At the same time, the hierarchical select adds $\Theta(\log Z)$ layers to the output ciphertext as opposed to a single layer. Clearly, this makes the layer bound from Theorem 2 increase to $\Theta(\log Z \log N)$, and we compensate by setting $s_0^{\mathsf{tree}} = \Theta(\log Z \log N)$ which gives $B_c^{\mathsf{tree}} = \Theta(\gamma \log \lambda \log N)$ for our choice of Z. Repeating the block size analysis with B_c^{tree}, the select vector size on reads becomes $\Theta(\gamma \log^2 \lambda \log N)$, on evictions becomes $\Theta(\gamma \log^2 \lambda \log^2 N)$ bits (amortized). So our final block size requirement is $B^{\mathsf{tree}} = \Omega(\gamma \log^2 \lambda \log^2 N)$, which was reported in Table 1.

5 Security Against Fully Malicious Server

So far, we have seen a scheme that achieves security against an *honest-but-curious* server who follows the scheme's specification correctly but wishes to learn information about the client's data and access pattern. 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 scheme's specification.

We start by describing several abstract properties of the Onion ORAM scheme from the previous section, and we will call any scheme with these properties an "abstract server-computation ORAM" scheme. We will show how to compile any abstract server-computation ORAM scheme that is secure in the honest-but-curious setting into a scheme secure in the fully malicious setting.

5.1 Abstract Server-Computation ORAM

Any semi-honest secure server-computation ORAM scheme with satisfying these properties is referred to as 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 $\mathsf{pt} = (\mathsf{pt}[1], \ldots, \mathsf{pt}[C])$ consists of C chunks of B_c bits each. An encrypted data block $\mathsf{ct} = (\mathsf{ct}[1], \ldots, \mathsf{ct}[C])$ consists of separate encryptions of each chunk. Let ct_i denote the encrypted data block in physical location i on the server, and pt_i denotes the underlying plaintext data. The client operates on the data blocks either by: (1) directly reading/writing an encrypted data block, or (2) instructing the server to apply the same function f to form a new data block ct_i, where $\mathsf{ct}_i[j]$ only depends on the j-th chunk of some 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 Onion ORAM scheme from the previous section is an instance 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 recursion [33]. The data blocks are encrypted under a layered additively homomorphic encryption scheme. Function f is a "homomorphic select operation", described by an encrypted select vector, 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.

Verifiable Metadata. We can use standard "memory checking" [2] schemes based on Merkle-Trees [26] to ensure that the client always reads the correct metadata, or aborts if the malicious server ever sends an incorrect value. A naive use of Merkle-Tree would add an $O(\log N)$ multiplicative overhead to the process of accessing the metadata, which is good enough for us. This $O(\log N)$ overhead can also be avoided by aligning the Merkle-tree structure with the ORAM-tree [30], or using generic authenticated data structures [27]. In any case, verifying metadata is basically free in Onion ORAM.

Verifiable Data Blocks: Initial Attempt. Unfortunately, we cannot rely on memory checking to protect the encrypted data blocks when the client doesn't read/write to them directly but rather instructs the server to compute on them. One simple attempt at adding verification would be to include a MAC for each plaintext data block as additional metadata to ensure "authenticity". Let's also assume the client has some mechanism to check freshness. Unfortunately, this is insufficient in the fully malicious setting. The problem is that a malicious server that learns *whether the client aborts or not* may learn some information about the client's data or access pattern.

Consider Onion ORAM for example where the malicious server wants to learn if, during a homomorphic selection operation, the location being selected is i. Then 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 in the above proposal depends on the decrypted value, which in turn 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 data block ciphertext and not on the plaintext data.

Verifiable Data Blocks: Actual Solution. For our actual solution, the client selects a random subset S consisting of λ chunk positions, which we call the "verification set". This set S is kept secret from the server. The client stores the subset of ciphertext chunks in positions $\{j : j \in S\}$ of every encrypted data block as additional metadata on the server, which we call the "verification chunks". That is, verification chunks are exact copies of ciphertext data blocks, λ chunks from each block. Verification chunks are additionally encrypted under the same scheme as the other metadata, 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 the verification set S against 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 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 "verification set" $S = \{s_1, \ldots, s_\lambda\} \subseteq [C']$. For each ciphertext data block ct_i in physical location i on the server, the client stores verification chunks $\mathsf{verCh}_i = (\mathsf{verCh}_i[1], \ldots, \mathsf{verCh}_i[\lambda])$ as additional metadata. 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. If the client detects a violation in metadata at any point, the client aborts (we call this abort_0).
– Whenever the client directly writes to an encrypted data block ct_i on the server, it also updates the corresponding verification chunks $\mathsf{verCh}_i[j] := \mathsf{ct}_i[s_j]$. Whenever the client instructs the server to update an encrypted data block ct_i using the aforementioned function f, the client applies the same function f on $\mathsf{verCh}_i[j]$ 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 stops and refuses to perform any future operations.

Theorem 3. *For any "abstract server-computation ORAM" scheme which is secure in the honest-but-curious setting, the above compiler yields a "verified server-computation ORAM" scheme which is secure in the fully malicious setting. We only assume the security of the memory-checking scheme, which can be achieved under collision-resistant hashing.*

Security Intuition. Notice that in the above scheme, the decision whether abort_1 occurs does not depend on any of the secret state of the underlying abstract server-computation ORAM scheme, and therefore we can reveal this information 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 locations in 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 C.3.

5.3 Verified Onion ORAM

We define Verified Onion ORAM to be the result of applying the above compiler to the Onion ORAM scheme from Section 4. The above theorem directly implies the following corollary.

Corollary 1. *Verified Onion ORAM achieves security in the fully malicious setting, assuming the security of the DCR assumption and the security of a memory-checking scheme (collision-resistant hashing).*

Setting the block size. We now re-apply the analysis from Section 4.5 to Verified Onion ORAM to get constant bandwidth blowup. We will use the hierarchical select from Section 4.5, thus the worst-case ciphertext chunk size is $B_c^{\mathsf{tree}} = \Theta(\gamma \log \lambda \log N)$ using the Damgård-Jurik cryptosystem. The main difference from semi-honest Onion ORAM is that on a read, the client must additionally download $\Theta(\lambda^2 \log N)$ verification chunks, or $\Theta(B_c^{\mathsf{tree}} \lambda^2 \log N) = \Theta(\gamma \lambda^2 \log \lambda \log^2 N)$ bits. ORAM evictions still transmit $\Theta(B_c^{\mathsf{tree}} \lambda \log N) = \Theta(\gamma \lambda \log \lambda \log^2 N)$ bits for select vectors (amortized). The error-correcting code makes each data block grow by only a constant factor. Thus, the block size we need to achieve constant bandwidth over the entire protocol is $B = \Omega(\gamma \lambda^2 \log \lambda \log^2 N)$.

Optimization: permuted buckets. Observe that the limiting factor for the block size in the above construction is the data transmitted on reads. We can reduce data movement on reads by a factor of λ with the following optimization used in Ring ORAM [31]: 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 (similar ideas are used in hierarchical ORAMs [17]). To avoid failure, 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 don't run out except with probability $2^{-\Theta(\lambda)}$. With this technique, the bandwidth cost on reads becomes $\Theta(B_c^{\mathsf{tree}} \lambda \log N) = \Theta(\gamma \lambda \log \lambda \log^2 N)$, bringing the block size to $B = \Omega(\gamma \lambda \log \lambda \log^2 N)$ as reported in Table 1.

6 Conclusion

This paper proposes *Onion ORAM*, the first concrete 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) and instead only requires an additive homomorphic scheme such as the Damgård-Jurik cryptosystem (based on Paillier's classical scheme). We further extend Onion ORAM to be secure in the fully malicious setting using standard assumptions. Due to the known efficiency of the types of cryptosystems our schemes require, we think of our work as an important step towards *practical* constant bandwidth blowup ORAM schemes.

References

1. D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In *Public-Key Cryptography–PKC 2014*, pages 131–148. Springer, 2014.
2. M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.
3. R. Canetti. Security and composition of multiparty cryptographic protocols. 13(1):143–202, 2000.
4. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. pages 136–145, 2001.
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 and C. Ravishankar. Combining oram with pir to minimize bandwidth costs. In *Proceedings of CODASPY*, 2015.
7. 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.
8. 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.
9. 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.
10. S. Garg, S. Lu, R. Ostrovsky, and A. Scafuro. Garbled RAM from one-way functions. Cryptology ePrint Archive, Report 2014/941, 2014. http://eprint.iacr.org/2014/941.
11. 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.
12. C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345, 2014.
13. C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled RAM revisited. pages 405–422, 2014.
14. C. Gentry, S. Halevi, M. Raykova, and D. Wichs. Outsourcing private RAM computation. pages 404–413, 2014.
15. C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. pages 99–108, 2011.
16. O. Goldreich. Towards a theory of software protection and simulation on oblivious rams. In *STOC*, 1987.
17. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. In *J. ACM*, 1996.
18. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.
19. M. Keller and P. Scholl. Efficient, oblivious data structures for mpc. Cryptology ePrint Archive, Report 2014/137, 2014. http://eprint.iacr.org/.

20. E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.
21. H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *ISC*, pages 314–328, 2005.
22. 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.
23. S. Lu and R. Ostrovsky. How to garble RAM programs. pages 719–734, 2013.
24. 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.
25. T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining oram and pir. In *Proceedings of NDSS*, 2014.
26. R. C. Merkle. Protocols for public key cryptography. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.
27. A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 411–423. ACM, 2014.
28. R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.
29. P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Eurocrypt*, pages 223–238, 1999.
30. 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.
31. 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/.
32. 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.
33. 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.
34. E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.
35. E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.
36. 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.
37. 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.
38. 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.
39. 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/.
40. 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.
41. P. Williams and R. Sion. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 293–304, New York, NY, USA, 2012. ACM.
42. 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.
43. 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.
44. 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/.

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 [4].

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. Our simulation-based security notion simultaneously embodies the following intuitive security notions: 1) the confidentiality of outsourced data; 2) the obliviousness of data accesses; and 3) the authenticity and freshness of data blocks fetched.

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 $\mathsf{op} := (\mathsf{read}, ind)$ or $\mathsf{op} := (\mathsf{write}, ind, \mathsf{data})$; 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 $D[ind]$ 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 $\mathsf{op} := (\mathsf{read}, ind)$ 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 [3], requiring that a real-world execution "simulate" an ideal-world (reactive) functionality $\mathcal{F}$. At an intuitive level, our definition captures the privacy and verifiability requirements for an honest client, in the presence of a malicious server.

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 $\mathsf{op} := (\mathsf{read}, ind)$ or $\mathsf{op} := (\mathsf{write}, ind, \mathsf{data})$ 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 $D[ind]$ to the client, and updates $D[ind] := \mathsf{data}$ accordingly if this is a write operation. The client then forwards $D[ind]$ 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*

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

This definition is simulation-based [3] where the client is honest, and the server is corrupted. (The client is never malicious in our setting.) The definition also simultaneously captures *privacy* and *verifiability*. Intuitively, 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 happens to cheat, the client can detect it and abort the protocol.

B Background: The Damgård-Jurik Cryptosystem

The Damgård-Jurik cryptosystem, a generalization of Paillier's cryptosystem [29], 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 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 $\mathbb{Z}_{n^{s_0+i}}$.

C Proofs

C.1 Pre-Onion ORAM: Bounding Overflows

We now give formal proofs to show that buckets do not overflow in Pre-Onion ORAM except with negligible probability.

Proof. (of Theorem 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 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 $\mathsf{EvictAlongPath}$ operation where b is on the eviction path is the m_1-th $\mathsf{EvictAlongPath}$ operation, and the $\mathsf{EvictAlongPath}$ operation where b is a sibling bucket is the m_2-th $\mathsf{EvictAlongPath}$ 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 $\mathsf{EvictAlongPath}$ 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[3] 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.

C.2 Onion ORAM: Bounding Layers of Encryption

To bound the layers of onion encryption, we can consider the following abstraction. Suppose all blocks in the tree have a layer associated with it. Further, the layer of a bucket is the maximum layer of any block in it.

- Blocks in the root bucket are 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{parent.layer, child.layer} + 1.

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

To prove tight bounds on the number of encryption layers, we will focus our attention on a single bucket, and consider all bucket-triplet operations that this bucket is involved in. This bucket will be involved in one of three roles, as the source bucket, as the sibling bucket, or as the destination bucket. We now state an important observation.

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

Proof. Straightforward from the definition of reverse lexicographical order.

The above observation is important in the following sense: it shows that each bucket will not be written into more than twice, before it is eviction. This observation, in combination with Observation 1 of Section 3 will allow us to bound the number of encryption layers.

Proof. (of Thereom 2). We prove by induction.

Base case. The theorem holds obiviously 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 a single bucket denoted 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.

[3] 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.

Based on Observation 1, after each source operation, bucket becomes empty. Therefore, it must hold that

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

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

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

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

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

Finally, our post-processing on leaves where the client peels of the onion layers extends this theorem 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 blocks at level k in the tree have at most $k + 1$ layers.

C.3 Malicious Security Proofs

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.