# Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM

Christopher Fletcher
MIT
cwfletch@mit.edu

Muhammad Naveed
Cornell/UIUC
naveed2@illinois.edu

Ling Ren
MIT
renling@mit.edu

Elaine Shi
Cornell
elaine@cs.cornell.edu

Emil Stefanov
Berkeley

**Abstract**

Known Oblivious RAM (ORAM) constructions achieve either optimal bandwidth blowup or optimal latency (as measured by online roundtrips), but not both. We are the first to demonstrate an ORAM scheme, called Bucket ORAM, which attains the best of both worlds. Bucket ORAM simultaneously achieves a single online roundtrip as well as constant overall bandwidth blowup.

## 1 Introduction

Oblivious RAM, initially proposed by Goldreich and Ostrovsky [18, 19, 32], is a useful cryptographic building block that allows a client to outsource data to an untrusted server, while hiding both data contents as well as access patterns from the untrusted server. Since the original work by Goldreich and Ostrovsky [18, 19, 32], many subsequent works have demonstrated improved constructions [3, 6, 8, 10–12, 20, 22, 24, 33, 34, 37, 41–46].

The cost of Oblivious RAM constructions can be characterized by several related but different metrics. Two of the most important metrics are

1. *Bandwidth blowup*, i.e., for the client to access a single block, how many blocks on avarage must be transmitted between the client and the server to hide the true block of intent. This metric accounts for both online and offline traffic, and hence is also referred to as *overall bandwidth blowup*. Bandwidth blowup is closely related to the throughput of the ORAM under maximum load [39].

2. Response time or *latency*, i.e., the minimum delay from a client's request till the block is retrieved. The second metric is important for characterizing the ORAM's user-perceived performance under real-life, bursty traffic [11], and is closely related to the online bandwidth and the number of roundtrips.

These cost metrics typically depend on how much private storage the client has. In this paper we focus on the small client storage case, assuming that the client can store up to (poly)-logarithmic number of blocks.

| Construction | Latency (RTs) | Client store (blocks) | BW blowup | Server I/O (blocks) | Block size (bits) |
|---|---|---|---|---|---|
| **Path ORAM** [41] | $O(\log N)$ | $O(\log N)\omega(1)$ | $O(\log N)$ | $O(\log N)$ | $\Omega(\log^2 N)$ |
| **SR-ORAM** [44] | 1 | $O(1)$ | $O(\log^2 N)$ | $O(\log^2 N)$ | $\omega(\log^2 N)$ |
| **Onion ORAM** [12] | $O(\log N)$ | $O(1)$ | $O(1)$ | $O(\log N)$ | $\widetilde{\Omega}(\log^6 N)$ |
| **Bucket ORAM** | 1 | $O(\log N)\omega(1)$ | $O(\log N)$ | $O(\log N)$ | $\omega(\log^3 N)$ |
| **Bucket ORAM** | 1 | $O(1)$ | $\widetilde{O}(\log N)$ | $\widetilde{O}(\log N)$ | $\omega(\frac{\log^3 N}{\log\log N})$ |
| **Bucket ORAM + AHE** | 1 | $O(1)$ | $O(1)$ | $O(\log N)$ | $\widetilde{\Omega}(\log^6 N)$ |

**Table 1: Comparison of Bucket ORAM and known constructions.** $\widetilde{O}$ and $\widetilde{\Omega}$ hides $\log\log N$ to $\mathsf{poly}(\log\log N)$ factors. Asymptotical costs listed here are for *malicious* security. The security of all schemes are parameterized to have neligible in $N$ failure probability. Server I/O counts the amortized number of blocks the server touches per access.

Recent works have endeavored to optimize each of the above metrics separately. On the bandwidth side, recent ORAM schemes [41, 42] achieved $O(\log N)$ bandwidth blowup, with the hidden constant also highly optimized to approach 2 [35]. More recently, ORAM schemes with *constant* bandwidth blowup have been constructed using either fully homomorphic encryption [3], or additively homomorphic encryption [12, 31]. Although on the surface these results seem to contradict the logarithmic lower bound on ORAM bandwidth blowup shown by Goldreich and Ostrovsky [18, 19, 32], it is now well-understood that this lower bound can be circumvented by leveraging server-side computation. On the other hand, to best optimize the latency metric, recent works [11, 44] showed how to construct ORAMs with a single *online roundtrip*, i.e., only a single client-server interaction is needed to fetch a block (not counting offline shuffling).

Among all the above works, however, the constructions with promising bandwidth results do not try to optimize roundtrips, while the constructions with single roundtrips incur at least $\Omega(\log^2 N)$ bandwidth blowup. Given such separate encouraging new progress, a natural question is whether we can achieve the best of both worlds, that is, *can we construct an ORAM scheme that simultaneously achieves optimal online roundtrips as well as optimal bandwidth blowup?*

**Our results and contributions.** We give a positive answer to the above challenge by constructing a new ORAM scheme called Bucket ORAM. Bucket ORAM (with the use of additively homomorphic encryption) achieves a single online roundtrip, and $O(1)$ bandwidth blowup. Without additively homomorphic encryption, Bucket ORAM has $\widetilde{O}(\log N)$ bandwidth blowup, still a significant improvement over SR-ORAM [44], the best existing construction with a single online roundtrip. We refer the reader to Table 1[1] for a more detailed comparison with the state-of-the-art ORAM schemes.

## 1.1 Technical Highlights

We carefully design the Bucket ORAM protocol to achieve bandwidth efficiency, and at the same time to be compatible with known techniques for both single online roundtrip as well as constant bandwidth blowup. In the process, we also uncover interesting new hybrid techniques in ORAM

[1]Throughout the paper, we say that a scheme's cost is $f(N) = O(g(N))\omega(1)$, iff for any $\alpha(N) = \omega(1)$, there exists a scheme whose cost is $O(g(N)\alpha(N))$.

construction that may be of independent interest: Bucket ORAM is a hybrid between two ORAM frameworks and enjoys nice properties of both.

Roughly speaking, two types of ORAM constructions have been proposed in the past, 1) those based on the hierarchical framework initially proposed by Goldreich and Ostrovsky [19]; and 2) those based on the tree-based framework initially proposed by Shi et al. [37].

Both hierarchical ORAMs and tree-based ORAMs share the common feature of having exponentially growing levels. One fundamental difference between the two frameworks is whether they impose restrictions on a block's location within a level, and how they treat *shuffling* of data blocks (also referred to as *eviction*).

**Global vs. local data shuffles.** Hierarchical ORAMs impose no restriction on a block's location within a level. A block can reside anywhere within the level and its location is typically computed by a pseudorandom function. Correspondingly, shuffling is performed *globally* on all data blocks within a level, and requires a global oblivious sort on all data blocks within the level. Each level is rebuilt periodically, and larger levels are rebuilt exponentially less often than smaller levels.

By contrast, tree-based ORAMs restrict a block's location to a tree path as the block travels down levels of the hierarchy. This allows tree-based schemes to rely on *local* evictions to perform data shuffles. Each atomic eviction operation typically touches only a small number of blocks (e.g., a path in the tree). One or a small number of such atomic evictions happen after each data access.

It is known that tree-based ORAMs' local eviction strategy helps improve bandwidth by avoiding oblivious sorts on a large number of blocks. Indeed, the most bandwidth efficient (both asymptotically and empirically) ORAMs so far are all tree-based [35, 41, 42].

On the other hand, the level-rebuild-style shuffling strategy of hierarchical ORAMs have certain desirable properties. In particular, Williams and Sion show a technique to construct a single online roundtrip ORAM [44], and their technique is only compatible with level-rebuild-style eviction, but not tree-based ORAM eviction algorithms. Besides this, level-rebuild-style eviction naturally achieves *steady progress*, i.e., blocks will not linger in smaller levels during eviction. In the Onion ORAM work, Devadas et al. [12] showed a technique that leverages additively homomorphic encryption to achieve constant bandwidth blowup, and their technique works only with ORAMs with steady progress. Level-rebuild-style shuffles also achieve *predictive time*, which has theoretical applications such as in the construction of (reusable) Garbled RAM schemes [15–17, 28].

**A hybrid between hierarchical ORAM and tree-based ORAM.** Bucket ORAM adopts the level-rebuild-style data shuffling of hierarchical ORAMs, where larger levels are rebuilt exponentially less often than smaller levels. At the same time, Bucket ORAM also places restrictions where blocks can reside in a level much as in tree-based ORAMs. In this way, Bucket ORAM's level rebuild eviction involves the client performing local operations on $O(1)$ number of buckets at a time, and there is no need to perform global oblivious sortings on entire levels.

Section 2 describes this basic hybrid ORAM construction. Later in Sections 3, we show that this hybrid ORAM construction can be further optimized to achieve the efficiency of tree-based ORAM. Further, since the construction preserves nice properties of the hierarchical ORAM such as *steady progress* and *predictive time*, we are able to adapt known techniques [12, 44] for making it both single round and constant bandwidth blowup (see Sections 4 and 5).

### 1.2 Related Work

Oblivious RAM was first proposed by Goldreich and Ostrovsky [18,19,32] who also proposed the well-known hierarchical framework for constructing ORAMs. Hierarchical constructions were subsequently improved in numerous works [20, 24, 33, 34, 43–46]. More recently, a new, tree-based framework for constructing ORAMs was proposed [37], and later improved in numerous works [8, 12, 35, 41, 42].

Classically, ORAM was formulated in a setting without server-side computation. In this setting, a logarithmic lower bound was shown by Goldreich and Ostrovsky [18,19,32]. Wang et al. show that this lower bound is in some sense tight in a setting without server-side computation [42]. Several recent works [3, 12, 30] show how to leverage server-side computation to achieve constant bandwidth blowup, circumventing this logarithmic lower bound. Notably, the techniques by Devadas et al. [12] rely on only additive homomorphic encryption (rather than fully-homomorphic encryption), and achieve malicious security with standard assumptions (without expensive SNARKs). Devadas et al. explain that to apply their techniques, the underlying ORAM must satisfy a property called "steady progress", requiring that blocks do not linger in smaller levels during eviction.

Williams and Sion recently proposed SR-ORAM, a small client-storage ORAM with a single online roundtrip [44]. Their construction is built atop the hierarchical framework, which naturally satisfies the "steady progress" property. Although with some careful work, it might be possible to adapt the constant bandwidth technique by Devadas et al. [12] to the SR-ORAM, it is apparent that doing so would result in at least $O(\log^2 N)$ server I/O cost. In comparison, our Bucket ORAM relies on a new hybrid ORAM and achieves $O(\log N)$ server I/O cost (see Table 1). Besides SR-ORAM, Burst ORAM [11] also attains single online roundtrip but requires storing a linear amount of metadata on the client side.

Numerous applications of ORAM have been proposed. For example, recent works have demonstrated the practicality of implementing ORAMs in secure processor architectures to defend against physical probing of memory and buses [13, 14, 25, 29, 36]. Numerous works explored implementing ORAM for privacy-preserving storage outsourcing [11, 39, 45, 46]. Oblivious RAM has also been implemented for RAM-model secure multiparty computation [2, 26, 27].

## 2 Basic Construction: A Hybrid between Tree-based and Hierarchical ORAM

**Bucket ORAM: level-rebuild-style, local shuffles.** Bucket ORAM features level-rebuild style shuffling just like in hierarchical ORAMs, but avoids having to perform oblivious sorts globally on all blocks within a level. Instead, blocks within a level are divided into $\omega(\log N)$-sized buckets, and the client works on four buckets at a time during the level rebuild.

We present the basic, interactive version of the Bucket ORAM algorithm below. Later in Section 3 and 4, we show how to achieve single online roundtrip.

### 2.1 Server- and Client-Side Data Structures

**Server layout.** Sever storage is organized into $L = O(\log N)$ *levels* where level $\ell$ contains $2^\ell$ buckets for $\ell \in \{0, 1, \ldots, L-1\}$. Each bucket stores $Z$ blocks, and each block is of $B$ bits in length. Each block is either a *real* block or a *dummy block*, and if the number of real blocks is $< Z$, we fill the rest of the bucket with dummies.

ORAM.Request(op, idx, data): *// Precondition:* ebuffer *is not full*

1: $\mathsf{label}^* \leftarrow \mathsf{UniformRandom}(\{0,1\}^{2^L-1})$
2: label ← position[idx], position[idx] ← label*
3: block ← ⊥
4: **for** each $\mathsf{block}_0 \in \{\mathsf{ebuffer} \cup \mathcal{P}(\mathsf{label})\}$ **do** *// path from leaf* label *to root*
5: **if** $\mathsf{block}_0.\mathsf{idx} = \mathsf{idx}$ **then**
6: $\mathsf{block} \leftarrow \mathsf{block}_0$
7: **end if**
8: **end for**
9: **If** op is "write": block.data ← data
10: block.label ← label*
11: ebuffer ← ebuffer ∪ {block}
12: **return** block.data

**Figure 1: Algorithms for the request phase (basic, interactive version).** bucket.ReadAndRemove scans through the bucket, fetches and removes a block identified by idx from a bucket. $\mathcal{P}(\mathsf{label})$ denotes the path from the root to the leaf node with label.

To help achieve local shuffles, we will treat the server storage as a *binary tree* where each bucket is a node in the tree. We get a tree structure by first labeling each bucket in level $\ell$ with a unique identifier $i \in \{0, 1, \ldots, 2^\ell - 1\}$. This way, each bucket in level $\ell < L - 1$ has two distinct child buckets in level $\ell + 1$, whose respective identifiers are $2i$ and $2i + 1$.

**Client storage.** Like in tree-based ORAM schemes, the client stores an $O(N \log N)$-bit *position map* that maps each block to a bucket in level $L - 1$. The client also stores an *eviction buffer* of size $Z' = Z/O(1)$ (denoted ebuffer) containing blocks that have not been evicted yet.

**Block metadata.** To enable all ORAM operations, each block carries with it some metadata besides it payload. For the purpose of this section, each block's metadata include its type (either "real" or "dummy"), and for a real block, its logical address idx and path label it's mapped to. Therefore, block formats are as below:

| | | | | |
|---|---|---|---|---|
| Real block: | ("real", | data, | idx, | label) |
| Dummy block: | ("dummy", | data = _, | idx = _, | label = _) |

In Sections 3 and 4 we will introduce additional metadata for blocks, as well as other per-level auxiliary metadata to enable single roundtrips.

## 2.2 Algorithm Invariants and Operations

**Main Invariant.** Like in tree-based ORAMs, every block is assigned a position label indicating a leaf node in the tree, and the mapping is stored in the position map. A block always resides along the path to its designated leaf node.

**Recursion.** The position map is the $O(N \log N)$-bit. The client can reduce this local storage to $O(1)$ bits by recursively storing the position map in other smaller ORAMs using the standard

ORAM.Eviction(block): *// happens every $Z'$ requests*

1: Pad ebuffer with $Z - Z'$ or more dummy blocks such that its size is $Z$
2: **if** AllLevels[0] is empty on server **then**
3: Write back ebuffer to AllLevels[0]
4: return
5: **end if**
6: Let level ← (ebuffer)
7: Let $\ell^*$ := first empty level or $L - 1$ if all levels are full
8: **for** $\ell = 0$ to $\ell^* - 1$ **do**
9: level ← Merge(level, AllLevels[$\ell$])
10: **end for**
11: **if** AllLevels[$\ell^*$] is empty **then**
12: AllLevels[$\ell^*$] := level
13: **else** *// $\ell^*$ must be $L - 1$*
14: AllLevels[$\ell^*$] := MergeInPlace(AllLevels[$\ell^*$], level)
15: **end if**

Merge(level, $\overline{\text{level}}$): *// level and $\overline{\text{level}}$ each contains $m$ buckets and resides on server*

1: Initialize an empty new $\text{level}_{\text{new}}$ of size $2m$ on server
2: **for** $i \in \{0, 1, \ldots, m - 1\}$ **do**
3: Fetch level[$i$] and $\overline{\text{level}}$[$i$] from server
4: $mid \leftarrow (2i + 1)N/2m$
5: $S_L \leftarrow \text{Left}(\text{level}[i], mid) \cup \text{Left}(\overline{\text{level}}[i], mid)$
6: $S_R \leftarrow \text{Right}(\text{level}[i], mid) \cup \text{Right}(\overline{\text{level}}[i], mid)$
7: $\text{bucket}_L \leftarrow \text{ConstructBucket}(S_L)$, $\text{bucket}_R \leftarrow \text{ConstructBucket}(S_R)$
8: Write back $\text{bucket}_L$ to $\text{level}_{\text{new}}[2i]$.
9: Write back $\text{bucket}_R$ to $\text{level}_{\text{new}}[2i + 1]$.
10: Return (a pointer to) $\text{level}_{\text{new}}$
11: **end for**

MergeInPlace(level, $\overline{\text{level}}$): *// level and $\overline{\text{level}}$ each contains $m$ buckets*

1: Initialize an empty new $\text{level}_{\text{new}}$ of size $m$ on server
2: **for** $i \in \{0, 1, \ldots, m - 1\}$ **do**
3: Fetch level[$i$] and $\overline{\text{level}}$[$i$] from server
4: bucket ← ConstructBucket(level[$i$] ∪ $\overline{\text{level}}$[$i$])
5: Write back bucket to $\text{level}_{\text{new}}[i]$.
6: **end for**

Left(bucket, $mid$):

1: Return {block ∈ bucket : (block is "real") ∧ (block.label < $mid$)}

Right(bucket, $mid$):

1: Return {(block ∈ bucket : (block is "real") ∧ (block.label ≥ $mid$)}

**Figure 2: Algorithms for eviction phase (basic, interactive version).** ConstructBucket($S$) pads $S$ with dummy blocks up to size $Z$ and returns the new bucket.

recursion technique [37, 41], Unfortunately, recursion increases the number of roundtrips. Later in Section 3, we will describe techniques to reduce client storage without resorting to recursion.

**Requesting a block.** In the basic Bucket ORAM construction, a client requests (either to read or to update) a block like in any tree-based ORAM (see Figure 1). To request a block with logical address $\mathsf{idx}$, the client looks up the position map to identify the path where the block resides. The client then reads every block in the eviction buffer and on the corresponding path. The client is guaranteed to find the block $\mathsf{idx}$ in the process. Finally, the requested block (possible updated by the client) is appended to the client's local eviction buffer.

**Eviction.** After $Z'$ requests, the client-side eviction buffer may be full and we will call an eviction routine (Figure 2). The goal of the eviction step is to write the blocks in the eviction buffer back to the server. We describe a novel eviction routine that achieves hierarchical ORAM's level rebuild using only local reshuffle. Therefore, we do not require large client storage or global oblivious sorting on a large set of blocks.

## 2.3 New Level Rebuild based on Local Shuffle

When the eviction buffer is full (contains $Z'$ blocks), an eviction takes place. An eviction causes a series of cascading $\mathsf{Merge}$ operations. The $\ell$-th $\mathsf{Merge}$ operation (where $0 \leq \ell < L-1$) merges two levels each containing $2^\ell$ buckets into a new level containing $2^{\ell+1}$ buckets. One of the two levels is a persistent copy corresponding to the $\ell$-th level in the server's main ORAM data structure, and the other copy is a transient one created during the previous cascading $\mathsf{Merge}$.

Just like in hierarchical ORAM scheme, the cascading merge behaves like a binary counter. Let $T$ denote the current time, where time is characterized by the number of *level-rebuild* operations thus far. Let $T := t_{L-1}t_{L-2}\ldots t_0 \mod 2^L$ denote the binary representation of $T$ modulo $2^L$ where $L$ is the number of levels. Then, $T$ defines the current state of all levels. $t_i = 1$ means that the $i$-th level is currently occupied, and $t_i = 0$ means that the $i$-th level is currently empty. Suppose that levels $0, 1, \ldots \ell - 1$ are consecutively full levels, and that $\ell < L$ is the first empty level. Then, an eviction operation would cause $\ell$ number of cascading merges until blocks get written to the $\ell$-th level.

If all levels are full, then the cascading merges stop at the largest level $L-1$. In other words, two levels of size $2^{L-1}$ will get merged into a level of size $2^{L-1}$, rather than a level of size $2^L$. This is OK because the largest level must have enough space for all the blocks in the ORAM.

**Quadruplet merge.** Suppose we need to merge two levels $\mathsf{level}, \overline{\mathsf{level}}$ each containing $m$ buckets into a larger level $\mathsf{level}_{\text{new}}$ with $2m$ buckets. We do this by operating on four buckets at a time as shown in Figure 3, henceforth referred to as the *quadruplet merge* algorithm. Buckets $\{\mathsf{level}_{\text{new}}[2i], \mathsf{level}_{\text{new}}[2i+1]\}$ are constructed from buckets $\{\mathsf{level}[i], \overline{\mathsf{level}}[i]\}$ by moving blocks one level up along their desired paths. In other words:

- The blocks from buckets $\mathsf{level}[i]$ and $\overline{\mathsf{level}}[i]$ that want to go *left* are placed into bucket $\mathsf{level}_{\text{new}}[2i]$.
- The blocks from buckets $\mathsf{level}[i]$ and $\overline{\mathsf{level}}[i]$ that want to go *right* are placed into bucket $\mathsf{level}_{\text{new}}[2i+1]$.

If the client has a transient buffer as large as the capacity of two buckets, the quadruplet merge algorithm can be performed by having the client download a pair of buckets $\{\mathsf{level}[i]$ and $\overline{\mathsf{level}}[i]\}$ at a time, rearrange the blocks locally as mentioned above to form $\mathsf{level}_{\text{new}}[2i]$ and $\mathsf{level}_{\text{new}}[2i+1]$ and then writing the new buckets back to the server.

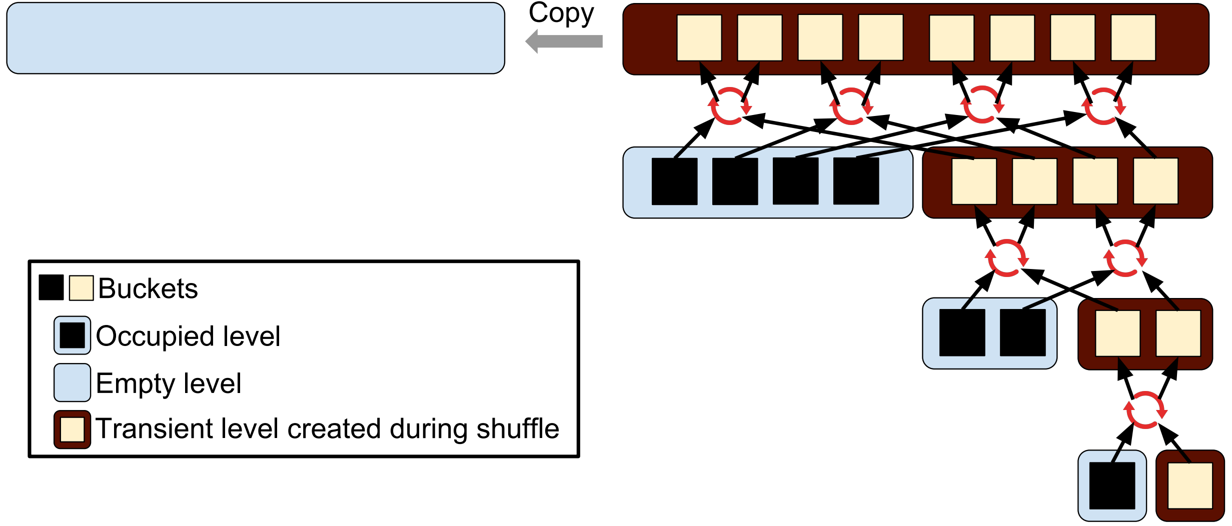



**Figure 3: Writing back a bucket to the server causes a sequence of cascading level merges.** Each level merge combines two levels of the same size into a bigger level of twice the size. Level merge is performed with a sequence of quadruplet merges, where the client operates on four buckets at a time.

For a client with constant local storage, the quadruplet merge algorithm can be performed using oblivious sorting, incurring an extra $O(\log Z)$ multiplicative bandwidth overhead.

## 2.4 Bounding Overflow and Cost Analysis

**Lemma 1.** *With bucket size $Z$, eviction buffer size $Z'$ and number of levels $L \geq \log_2(2N/Z')$, the probability of any bucket overflowing is at most $e^{-\frac{Z-Z'}{3Z'}}$.*

*Proof.* Level $\ell$ is rebuilt every $Z' \cdot 2^\ell$ accesses. There are at most $2^\ell Z'$ incoming blocks, and each incoming block makes an independent choice in one of the $2^\ell$ buckets. A simple Chernoff bound gives the bound. □

Following most prior works, we will set $Z = 2Z' = O(\log N)\omega(1)$, giving $N^{-\omega(1)}$ overflow probability (negligible in $N$). We can also set $Z = 2Z' = \Theta(\lambda)$ if one needs overflow probability exponentially small in the security parameter $\lambda$.

Our basic Bucket ORAM algorithm described in this section has the following cost profile.

**Client storage.** The position map is $N \cdot \log N$ bits in size, but can be recursively stored until it becomes $O(1)$ in size. A transient storage of $O(1)$ buckets ($O(Z)$ buckets) is required to perform the quadruplet merge operations during level rebuilding. Therefore, overall, the client storage is $O(\log N)\omega(1)$ blocks.

**Online bandwidth.** To request a block, the client reads an entire path containing $ZL = O(\log^2 N)\omega(1)$ blocks, so the online bandwidth blowup is $O(\log^2 N)\omega(1)$.

We remark that the permuted bucket technique from Ring ORAM [35] can reduce the online cost to $O(\log N)$ (or even $O(1)$ if server computation is allowed), but at the expense of a constant factor number of additional roundtrips. We will not go into any details of those techniques as they are not needed in our main constructions in later sections.

**Offline bandwidth.** Rebuilding a level of with $n$ blocks requires reading and writing $O(n)$ blocks. It is not hard to see that the amortized offline bandwidth blowup is $O(\log N)$ since a level containing $Z \cdot 2^l$ slots is evicted to once every $Z' \cdot 2^l$ requests where $Z' = \Theta(Z)$.

**Comparison to hierarchical ORAM schemes.** In a traditional hierarchical ORAM scheme, rebulding a level containing $n$ blocks involves an oblivious sort on all data blocks within the level. This requires reading and writing $O(n \log n)$ blocks. Our quadruplet merge algorithm asymptotically reduces the level rebuild cost with only $\Theta(Z) = O(\log N)\omega(1)$ blocks of transient client storage.

As mentioned earlier, in case the client has only $O(1)$ storage, our quadruplet merge algorithm can also be achieved by performing oblivious sorts on $O(Z)$ blocks at a time, incurring an extra $O(\log \log N)$ multiplicative factor in offline bandwidth.

# 3 Improved Construction with No Position Map

The standard recursion techniques [12,37,40–42] in the basic construction requires multiple $O(\log N)$ interactions with the server for the online phase. In this section, we describe techniques that will allow us to get rid of the client-side position map (hence recursion is not needed).

**Clarifications.** From this section onward, our techniques require that the server perform computation. Therefore, strictly speaking, we are in the setting of (Verifiable) Oblivious Storage as phrased by Apon et al. [3]. Many ORAM schemes [11, 12, 35], including SR-ORAM [44] also leveraged server computation without explicitly mentioning the change of model. In comparison, classical ORAM [18, 19, 32] assumes a setting where the server is a passive storage device that only supports read and write requests.

## 3.1 Intuition

**Real, mask, and dummy blocks.** In our new scheme, we develop techniques so that the client can read exactly one block per level (thereby improving online bandwidth) yet be able to execute the entire request operation in a single roundtrip. For this purpose, we need each block to be one of three types, a *real* block, a *mask* block, and a *dummy* block. A dummy block signals an empty slot in the level data structure. Real blocks are the same as before. However, here we additionally introduce a new type of block called a *mask block*. Upon being rebuilt, a level will have $Z' \cdot 2^\ell$ mask blocks (as many as there are real blocks), and every mask block has a counter $\mathsf{maskcnt} \in \{1, 2, \ldots, Z' \cdot 2^\ell\}$. The format of different blocks are summarized below where _ denotes "don't care" fields for each type of block:

| | | | | | |
|---|---|---|---|---|---|
| Real block: | ("real", | data, | idx, | label, | maskcnt = _) |
| Mask block: | ("mask", | data = _, | idx = _, | label = _, | maskcnt) |
| Dummy block: | ("dummy", | data = _, | idx = _, | label = _, | maskcnt = _) |

As in hierarchical ORAM and Ring ORAM [35], during each request the client will download the requested (real) block from a level if it is present, or a previously un-read mask block otherwise. For

security, the distribution of the mask blocks within the level must be identical to the distribution of real blocks in the level. And obviously each level must be rebuilt before all the mask blocks can possibly be consumed. Dummy blocks simply fill in the remaining empty spaces.

**Server stores blocks in a hash table.** To find the real block (or next mask block) in each level, we leverage a server-side hash table technique first proposed by Williams and Sion [44, 45]. For every (encrypted) block stored on the server, either real, mask, or dummy, the client uses a pseudo-random function to compute a hash key (denoted $\mathsf{hkey}$ henceforth), and reveals $\mathsf{hkey}$ to the server. The server builds a hash table over the blocks, such that blocks can be looked up by their $\mathsf{hkey}$. Now, to read a block from a level, the client would perform a sequence of actions at the end of which produces the $\mathsf{hkey}$ of the block to be retrieved. Specifically, if the block to be retrieved is within the current level, the client computes the real $\mathsf{hkey}$ of the block. If the block is not in the current level, the client would compute the $\mathsf{hkey}$ for the next mask block. As in the previous paragraph, for security we guarantee that *the client will ask for each* $\mathsf{hkey}$ *only once before the hash table is rebuilt.*

The client now reveals the $\mathsf{hkey}$ to the server to fetch this block. In this way, the client only fetches a single block from each level instead of $Z$. For security, it is important that $\mathsf{hkey}$ be pseudorandom and time-dependent. In other words, the same block will have a different $\mathsf{hkey}$ when it is written to the server again in the future.

**Bloom filter.** Accompanying the hash table trick, we also adopt a standard Bloom filter trick first introduced by Williams and Sion [44, 45]. To allow the client to determine whether a level contains the desired block, each level has a auxiliary (encrypted) Bloom filter. To look up a block in a level, the client looks up $k$ locations in the the Bloom filter first to determine if the level contains the desired block. For security, if the desired block has been found in smaller levels, the client looks up $k$ random locations in the Bloom filter. Otherwise, it looks up $k$ real locations.

**Oblivious sorts on metadata during level rebuilding.** During a level rebuilding, the client performs $O(1)$ number of oblivious sorts on metadata to achieve the following goals:

- Rebuild the encrypted Bloom filter in synchrony with the new level being rebuilt.
- Randomly distribute mask blocks to all buckets in a level, and assign unique mask counters.

Note that we will not need to oblivious sort data blocks at any point.

### 3.2 Detailed Algorithm for the Requests (Online Phase)

**Notations.** We use $\mathsf{sk}$ to denote a client secret key that is kept confidential from the server. We use $\mathsf{PRF}_{\text{subscript}}$ to denote different pseudo-random functions employed by the client.

**Pseudo-random hash keys for blocks.** Level $\ell$ has $Z' \cdot 2^\ell$ mask blocks, and every mask block has a unique mask counter denoted $\mathsf{maskcnt} \in \{1, \ldots, Z' \cdot 2^\ell\}$. This guarantees that each level has at least as many mask blocks as there are real blocks. As will be obvious later, for security, the mask counters for all mask blocks within a level must be (pseudo-)randomly distributed to buckets.

Then the every block has a hash key defined as follows:

$$\mathsf{hkey}(\mathsf{block}) := \begin{cases} \mathsf{PRF}_{\mathrm{H}}(\mathsf{sk}, T || \mathsf{block.idx}) & \text{if real } \mathsf{block} \\ \mathsf{PRF}_{\mathrm{H}}(\mathsf{sk}, T || \text{“mask”} || \mathsf{maskcnt}) & \text{if mask } \mathsf{block} \\ \text{random string} & \text{o.w.} \end{cases}$$

For each level $\ell$, the client stores a current mask counter denoted $\mathsf{maskcnt}_\ell$. When a level $\ell$ is rebuilt, its $\mathsf{maskcnt}_\ell$ is reset to 0.
To look up a block $\mathsf{idx}$, the client does the following:

Initialize $\mathsf{found} := \mathsf{false}$. Next, for each level $\ell = 0$ to $L-1$:

**Look up Bloom filter:**

- If not $\mathsf{found}$, look up $k$ real locations in the this level's Bloom filter, where location $i$ is computed as in Equation (1). If all $k$ locations are 1, conclude that the block is in this level, and set $\mathsf{found} := \mathsf{true}$.
- Else if $\mathsf{found}$, look up $k$ random locations in this level's Bloom filter.

**Compute and reveal hash key to server:**

- If block $\mathsf{idx}$ is in level $\ell$, then reveal the real $\mathsf{hkey} := \mathsf{PRF}_{\mathrm{H}}(\mathsf{sk}, T||\mathsf{block.idx})$ to the server, where $T$ denotes the last time level $\ell$ was rebuilt.
- If block $\mathsf{idx}$ is not level $\ell$, reveal a mask $\mathsf{hkey} := \mathsf{PRF}_{\mathrm{H}}(\mathsf{sk}, T||\text{“mask”}||\mathsf{maskcnt}_\ell)$ to the server, and then increment $\mathsf{maskcnt}_\ell$.

**Retrieve block:**

- The server now returns the $\mathsf{block}$ with the specified $\mathsf{hkey}$ to the client (along with the $\mathsf{block}$'s location within the level). The online phase ends here.
- In an *offline* roundtrip, the client marks the block fetched invalid, by setting $\mathsf{block.type} :=$ “dummy”.

**Figure 4:** New block access algorithm that fetches only one block per-level.

where $T$ denotes the time step when the level was rebuilt. We note that the level number can be uniquely inferred by the timestamp $T$, and therefore we do not separately include the level number in the PRF's input.

**Leverage a per-level Bloom filter to compute the hkey of block to fetch.** To allow the client to efficiently query whether a block is contained in a specific level, the client stores an auxiliary data structure, an encrypted Bloom filter on the server. To check if a block is within a specific level, the client computes $k$ locations in the Bloom filter as

$$\mathsf{loc}_i := \mathsf{PRF}_{\mathrm{BF}}(\mathsf{sk}, T||\mathsf{block.idx}||i) \tag{1}$$

where $1 \leq i \leq k$, and $T$ denotes the last time this level was rebuilt. However, the moment that a block is found in a level, for all future levels, the client looks up $k$ random locations instead. See Figure 4.

**Algorithm for requesting a block.** Putting everything together, the new algorithm for requesting a block is described in Figure 4. In this algorithm, the client fetches only one block per-level during a request.

**Input:** A level containing $m$ buckets, where each bucket contains $Z$ blocks, either real or dummy (there is no mask block in a newly built level after the quadruplet merge step).

**Output:** A new level, where real blocks reside at random locations in the same bucket. Let $n_1, \ldots, n_{2^\ell}$ be random variables denoting the number of balls in each bin when we throw $Z' \cdot 2^\ell$ balls into $2^\ell$ bins. Each bucket $i \in [2^\ell]$ has exactly $n_i$ mask blocks residing at random locations, where each mask block has a unique mask counter $\mathsf{maskcnt} \in \{1, 2, \ldots, Z' \cdot 2^\ell\}$. The remainder empty slots in each bucket are populated with dummy blocks.

**Algorithm:**

1. **Metadata array creation.** Create an array containing all blocks' metadata. Make a linear scan over the level and for each bucket $i = 0$ to $2^\ell - 1$, append (“dummy”, _, $i$) for each dummy block and (“real”, _, $i$) for each real block. Then, for $j = 1$ to $Z' \cdot 2^\ell$: pick a random bucket $i$ within this level, and append the metadata entry (“mask”, $j, r$) to the array. The tuple stipulates the “mask” block with $\mathsf{maskcnt} = j$ will end up in bucket $r$. Note that the number of metadata entries assigned to each bucket is guaranteed to be $\geq Z$ at this point.
2. **O-sort metadata.** Oblivious sort the above array based on increasing order of bucket number (the last element in the tuple). For the same bucket number, place real block before mask blocks before dummy blocks.
3. **Linear scan.** Make a linear scan over the array, and for each bucket number: preserve the first $Z$ entries and rewrite all remaining entries as (“dummy”, _, $\infty$).
4. **O-sort metadata.** Oblivious sort the above array based on increasing order of bucket number. When sorted, preserve the first $Z \cdot 2^\ell$ entries and discard the remainder of the array.
5. **Permute blocks within each bucket.** One bucket at a time, read the next $Z$ entries from the array (metadata for real, dummy and mask blocks) and all $Z$ data blocks in the bucket (some of which will be real blocks). Randomly permute blocks and metadata within the bucket (on the client side), and write the bucket back.

**Figure 5: Randomly distribute mask blocks to buckets during level rebuilding.** Each real block within a level is randomly assigned to a bucket, and placed in a random empty slot in the bucket. This procedure makes sure that each mask block is also randomly assigned to a bucket, and then placed in a random empty slot within the bucket. In this way, no matter whether a block is within the current level, the client always reads a random bucket and a random unread location within the bucket.

### 3.3 Detailed Algorithm for Eviction/Level-Rebuild

The algorithm for evicting back blocks to the server and rebuilding levels proceeds in a similar fashion as in Section 2.3. However, now the client has to additionally 1) obliviously rebuild metadata for this level; and 2) randomly distribute mask blocks to buckets, and assign a unique mask counter $\mathsf{maskcnt}$ from the range $\{1, 2, \ldots, Z' \cdot 2^\ell\}$ to each mask block. The random redistribution of mask blocks to buckets is required to ensure the mask blocks in the rebuilt level have the same distribution

**Input:** The metadata of all real, mask, and dummy blocks within a level, residing on the server side.

**Output:** A Bloom filter for this level, residing on the server side.

**Algorithm:**

1. **Initialization.** Make a linear scan over the metadata. For each block:
   - If block is real: create $k$ pairs (on server): $\{(\mathsf{loc}_i, 1)\}_{i\in[k]}$ where $\mathsf{loc}_i$ (Equation (1)) denotes a location in the Bloom filter that should be set to 1.
   - Else if block is mask or dummy: create $k$ pairs (on server): $\{(\perp,\perp),\ldots,(\perp,\perp)\}$.
2. **Padding.** Let $m$ be the total number of blocks (all three types) in a level. At the end of the last step, we have an array containing $k \cdot m$ pairs. Pad this array with the pairs $(1,\perp),(2,\perp),\ldots,(\mathsf{BFSize},\perp)$, where $\mathsf{BFSize}$ is the Bloom filter size.
3. **O-sort array.** Oblivious sort the padded array lexigraphically where $\perp$ is is considered lexigraphically larger than everything else. In the sorted array, all entries $(i, _)$ appear before $(j, _)$ if $i > j$ where $_$ denotes wildcard. Further, all $(i, 1)$ pairs appear before the $(i,\perp)$ pair. All $(\perp,\perp)$ pairs appear at the end.
4. **Deduplicate pairs.** In a linear scan, preserve only the first occurence of each $(i, _)$, and rewrite all other occurrences as $(\perp,\perp)$.
5. **O-sort array.** Oblivious sort the resulting array lexigraphically. The sorted result should be of the format
   $$(1, _),(2, _),\ldots,(m, _),(\perp,\perp),\ldots,(\perp,\perp)$$
   where each $(i, _)$ is either $(i, 1)$ denoting that the $i$-th bit of the Bloom filter should be set, or $(i,\perp)$ denoting that the $i$-th bit of the Bloom filter should be clear.
6. **Finalize Bloom filter.** In a synchronized scan of the above array and the Bloom filter, sequentially set all bits of the Bloom filter as indicated.

**Figure 6:** Obliviously rebuild Bloom filter in synchrony with a newly rebuilt level.

as real blocks from the server's perspective.

We note that only the rebuilt level requires the above two steps, i.e., rebuilding of metadata and redistribution of mask blocks. Intermediate levels created during the cascading merge will be empty after the eviction completes and will not be touched during request operations until they themselves are rebuilt. At a high level, the new level rebuild algorithm works as follows.

**Quadruplet merges on data blocks.** First, the client performs quadruplet merges as in Section 2.3 (i.e., perform all of $\mathsf{ORAM.Eviction}()$ in Figure 2). At this moment, observe that 1) the $\mathsf{hkeys}$ of the blocks have not been revealed to the server; and 2) the rebuilt level contains only real and dummy blocks (i.e., no mask blocks).

**Assign mask counters, and randomly distribute mask blocks to buckets.** When a rebuilt level is being accessed in the future, it is crucial for security that from the server's perspective, every real or mask block is assigned to a random bucket, and then a random location within the bucket. Real blocks that get merged in the level have random leaf labels whose values have not been revealed to the server earlier. And this guarantees that every real block is residing in a random bucket from the server's perspective. We wish for mask blocks to have this same distribution over the buckets in the level. Every mask block must also be assigned a unique $\mathsf{maskcnt}$ from a contiguous range $\{1, 2, \ldots, Z' \cdot 2^{\ell}\}$.

We describe an oblivious procedure for achieving the above in Figure 5. This algorithm relies on $O(1)$ oblivious sorting operations on *metadata* to distribute all mask blocks to random buckets.

Finally, at the end of this step, $\mathsf{hkeys}$ of blocks are revealed sequentially to the server.

**Rebuild Bloom filter.** Whenever a level is being rebuilt, the client rebuilds the level's Bloom filter in synchrony with the new level. Doing so would require $O(1)$ oblivious sorts on metadata only as shown in Figure 6. This algorithm is standard and a similar version was described by William and Sion [44, 45].

## 3.4 Parameter Choices and Cost Analysis

**Bucket size.** Originally, each bucket contains on average $Z'$ real blocks, but needs $Z$ slots to avoid overflow. Our new construction must guarantee that the sum of real and mask blocks per bucket does not exceed the bucket capacity $Z$. We accomplish this by decreasing $Z'$ relative to $Z$. Since the number of mask blocks per level is set to the maximum number of real blocks per level, the analyis from Section 2.4 tells us $Z'$ only needs to decrease by a constant factor for this purpose.

**Bloom filter parameterization.** We need to parameterize the Bloom filter to have negligible security failures. Based on the well known Bloom filter analysis [1], for a Bloom filter containing $n$ elements, we can let the Bloom filter size $\mathsf{BFsize} := O(n\lambda)$ and let the number of hashes $k := O(\lambda)$ to obtain a $2^{-O(\lambda)}$ security failure probability.

**Online cost.** It is not hard to see that to request a block, the client reads one block per level, i.e., a total of $O(\log N)$ blocks. Further, the client reads $k$ encrypted Bloom filter bits per level, and sends one $\mathsf{hkey}$ to the server per level. The total cost of metadata is therefore $O(\lambda \log N)$ to attain a security failure of $2^{-O(\lambda)}$.

**Offline cost.** Assume that the client can store a constant number of buckets. To shuffle a level of size $n$, the client and the server need to transmit $O(n)$ blocks in between. The Bloom filter creation requires $O(1)$ oblivious sorts on $O(\lambda n)$ words, each $O(\log(\lambda N))$ bits in length. Other metadata

(aside from the Bloom filter) are $O(\log n)$ bits per block. Oblivious sorts on these metadata (e.g., for distributing mask blocks) consume $O(n \log^2 n)$ bits of bandwidth, which is dominated by the cost of Bloom filter creation. Therefore, rebuilding a level of size $n$ requires transmitting $O(n)$ blocks and $O(\lambda n \log^2(\lambda N))$ bits of metadata. The amortized offline shuffling cost is therefore $O(\log N)$ blocks and $O(\lambda \log N \log^2(\lambda N))$ bits of metadata.

**Block size for attaining $O(\log N)$ bandwidth blowup.** To attain a security failure negligible in $N$, we let $\lambda := O(\log N)\omega(1)$. In this case, when the block size is $\omega(\log^3 N)$, the cost of metadata gets absorbed, and the overall bandwidth blowup would be $O(\log N)$.

## 3.5 Obliviousness

**Lemma 2** (Distribution of block locations within a level)**.** *From the server's perspective, after a level is rebuilt, every real and mask block resides in an independent, random bucket.*

*Proof.* First, every real block is assigned a fresh, random leaf $\mathsf{label}$ when the block was last fetched. This random choice of leaf $\mathsf{label}$ is kept hidden from the server until the block is next requested. This leaf $\mathsf{label}$ places a real block in a random bucket within the level. Second, mask blocks are distributed to random buckets during the level rebuild procedure. □

**Lemma 3** (Every $\mathsf{hkey}$ is fetched only once from the server)**.** *Every hash key* $\mathsf{hkey}$ *for a real or mask block is fetched only once by the server before the level is rebuilt. Further, the request phase will not run out of mask blocks, i.e., will not attempt to read a mask block whose* $\mathsf{maskcnt}$ *is greater than the number of mask blocks within the level.*

*Proof.* First, the mask counter for a given level is always incremented whenever a mask block is fetched, such that the next time, the client would be trying to fetch the next mask block. Second, when a real block is fetched from a level $\ell$, it is logically removed from the level $\ell$. Until the next time the level $\ell$ is rebuilt, the same block will be found in a level $\ell' < \ell$. Therefore, the next time the client seeks the same real block, it will have been found in a smaller level $\ell'$, and the client would be reading the next mask block from level $\ell$.

As mentioned earlier, each level $\ell$ is rebuilt with $Z' \cdot 2^\ell$ mask blocks. We also know that level $\ell$ will be rebuilt every $Z' \cdot 2^\ell$ time steps. Therefore, the mask blocks will not run out before the next rebuild. When the next rebuild happens, the level's mask counter is reset to 0, and all blocks within the level obtain fresh new $\mathsf{hkey}$s. since the $\mathsf{hkey}$s are time-dependent. □

**Lemma 4** (Bloom filter reads are oblivious)**.** *From the perspective of the server, the client reads $k$ independent, (pseudo-)random locations in the Bloom filter every time.*

*Proof.* If a block has been found in a smaller level, the client reads $k$ fresh, random locations in a level's Bloom filter. If the block has not been found in smaller levels, the client reads $k$ real locations computed with a pseudorandom function $\mathsf{PRF}_{\mathrm{BF}}$. Assuming security of the $\mathsf{PRF}_{\mathrm{BF}}$, we can pretend that these are random locations, and further it is important to observe that these locations have not been disclosed to the server before. In particular, observe that if the client reads $k$ real locations in level $\ell$'s Bloom filter looking for a specific block $\mathsf{idx}$. This means that block $\mathsf{idx}$ resides in a level $\ell' \geq \ell$. At the end of the present read operation, the block $\mathsf{idx}$ will be relocated to a level $\ell^* \leq \ell$. Until the next time level $\ell$ is rebuilt, block $\mathsf{idx}$ will always exist in a smaller level than $\ell$. This means that the client will never look for block $\mathsf{idx}$ in level $\ell$ again till the next time level $\ell$ is rebuilt. As a

result, when the client discloses the real Bloom filter locations to the server, these locations cannot have been disclosed before. ☐

**Lemma 5** (Offline shuffling is oblivious)**.** *The level rebuilding algorithm has deterministic, predictable access patterns.*

*Proof.* Straightforward from the description of the algorithms. ☐

**Theorem 1** (Obliviousness of the ORAM scheme)**.** *Assume that the symmetric-key encryption scheme used to encrypt data and metadata satisfies semantic security and that* $\mathsf{PRF}_H$ *and* $\mathsf{PRF}_{BF}$ *are secure pseudorandom functions. Then, the ORAM scheme described in this section satisfies semi-honest security (i.e., replace Definition 2 of Appendix B with a semi-honest adversary).*

*Proof.* Based on the set of lemmas above, it is straightforward to construct an ideal-world simulator for a semi-honest, real-world adversary. Since the symmetric-key encryption scheme is semantically secure, the simulator simulates all ciphertexts by random encryptions of 0 of appropriate length. The time step $T$ and the occupied/empty status of each level is known by the simulator, and the access patterns of the offline shuffling is deterministic and predictable. Therefore, the simulator can easily simulate the offline shuffling. For the online request phase, the simulator simulates by reading $k$ random locations in the Bloom filter in each level. The simulator also discloses the $\mathsf{hkey}$ of a random block in this level to the real-world adversary. Each block's $\mathsf{hkey}$ is simulated by picking a fresh random string of appropriate length. It is not hard to argue that no polynomial-time environment can distinguish the real- and the ideal-worlds. ☐

### 3.6 Malicious Security

Due to lack of space, we give our construction for malicious security in Appendix C. We remark that it uses a combination of standard MAC and merkle tree techniques.

## 4 Making it Single Online Roundtrip

We now demonstrate how to achieve single online roundtrip. The idea is to rely on a garbling scheme for layered branching programs. We observe that the online phase of the block access algorithm (Figure 4) can be expressed efficiently as a layered branching program parameterized by the block $\mathsf{idx}$ to be accessed as well as the current time $T$. The inputs to this branching program are Bloom filter bits.

Previously, the online phase of the block access algorithm (Figure 4) was evaluated by the client, fetching data from the server as necessary — and thus introducing multiple roundtrips. A garbling scheme will allow the client to outsource this work entirely to the server. Basically, the client garbles both the Bloom filter bits as well as the branching program and send the garbled versions to the server. Now, the server evaluates the block access algorithm (Figure 4) by itself, without help from the client.

We note that Williams and Sion [44] construct a customized garbling scheme for a special branching program similar to Figure 4. However, their treatment of garbled branching programs is not formal or general. In the subsections below, we first give a generic abstraction and formal definition for garbled branching programs. We then show how to express the block access algorithm (Figure 4) as a layered branching program such that we can apply the garbling scheme.

## 4.1 Definitions for Branching Programs

**Branching program.** We consider a standard definition of a branching program. A branching program denoted $\mathsf{bp}$ is a fanout-2 directed acyclic graph with a single source and a number of sinks, and where each non-sink node is labeled a variable from an input string $x := (x_1, \ldots, x_m) \in \{0,1\}^m$. Every node $v \in \mathsf{bp}$ can be labeled with some payload string $v.\mathsf{payload} \in \{0,1\}^*$. The two edges going out of each internal node are labeled with 0 and 1 respectively. Computation starts from the source node. In each step, the branching program looks a bit in the input sequence $x = (x_1, \ldots, x_m)$ and based on its value, decides which edge to follow. Let $(v_1, \ldots, v_t)$ denote the sequence of nodes encountered when evaluating $\mathsf{bp}$ on input $x$. Finally, the branching program outputs:

$$\mathsf{bp}(x) := (v_1.\mathsf{payload},\ \ v_2.\mathsf{payload},\ \ldots,\ v_t.\mathsf{payload})$$

**Layered branching program.** A layered branching program is a branching program, but the nodes of the graph form layers such that the source is at layer 0, and the sinks are in the final layer. Nodes from layer $i-1$ have outgoing edges only into layer $i$. Henceforth, we will focus our discussions on layered branching programs. We refer to the number of layers as the *depth* of the layered branching program, and the number of nodes in each layer as the *width* of the layered branching program.

**Trace function.** We would like to define a garbling scheme for a layered branching program $\mathsf{bp}$ and some input $x$, such that an evaluator can evaluate the garbled $\mathsf{bp}$ on the garbled input, leaking only some trace and nothing else. Below we define this trace function.

Let $\mathcal{BP}[m, n_1, n_2, \ldots, n_D]$ denote the family of layered branching programs that computes on inputs of $m$ bits, contains $D$ layers, and where each layer $i$ has $n_i$ nodes. Given a layered branching program $\mathsf{bp} \in \mathcal{BP}[m, n_1, n_2, \ldots, n_D]$, and some input $x := (x_1, x_2, \ldots, x_m)$, Let $v_i$ denotes the node in layer $i$ encountered on the evaluation path for input $x$. We now define the trace function to contain the locations of the input bits looked at on the evaluation path, as well as the outcome $\mathsf{bp}(x)$.

$$\mathsf{tr}(\mathsf{bp}, x) := (v_1.\mathsf{addr}, v_2.\mathsf{addr}, \ldots, v_D.\mathsf{addr}, \mathsf{bp}(x))$$

## 4.2 Garbling Scheme for Layered Branching Programs

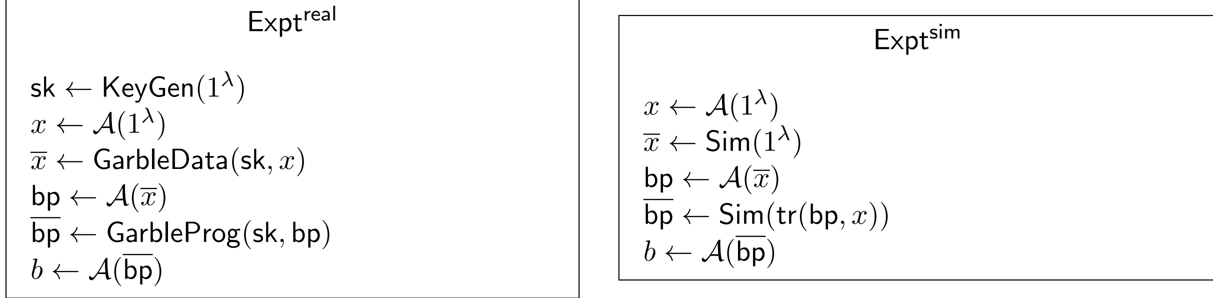
A garbling scheme for the family $\mathcal{BP}[m, n_1, n_2, \ldots, n_D]$ contains the following algorithms:

- $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$: On input $1^\lambda$, generates a secret key $\mathsf{sk}$.
- $\overline{x} \leftarrow \mathsf{GarbleData}(\mathsf{sk}, x)$: Given secret key $\mathsf{sk}$ and input $x \in \{0,1\}^m$, outputs the garbled input $\overline{x}$.
- $\overline{\mathsf{bp}} \leftarrow \mathsf{GarbleProg}(\mathsf{sk}, \mathsf{bp})$: Given a secret key $\mathsf{sk}$ and a layered branching program $\mathsf{bp} \in \mathcal{BP}[m, n_1, \ldots, n_D]$, outputs a garbled branching program $\overline{\mathsf{bp}}$.
- $\mathsf{res} \leftarrow \mathsf{Eval}(\overline{\mathsf{bp}}, \overline{x})$: Given garbled $\overline{\mathsf{bp}}$, and garbled input $\overline{x}$, evaluates the outcome $\mathsf{res}$.

**Correctness.** Correctness is defined in the obvious manner. We say that a garbling scheme for the family $\mathcal{BP}[m, n_1, \ldots, n_D]$ is correct, if for any $\mathsf{bp} \in \mathcal{BP}[m, n_1, \ldots, n_D]$, and for any $x \in \{0,1\}^m$, we have that

$$\Pr\left[\begin{array}{l} \mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda), \\ \overline{x} \leftarrow \mathsf{GarbleData}(\mathsf{sk}, x), \\ \overline{\mathsf{bp}} \leftarrow \mathsf{GarbleProg}(\mathsf{sk}, \mathsf{bp}) \end{array} : \mathsf{Eval}(\overline{\mathsf{bp}}, \overline{x}) = \mathsf{bp}(x)\right] = 1$$

**Security.** We say that a garbling scheme for the layered branching program family $\mathcal{BP}[m, n_1, \ldots, n_D]$ is secure, if there exists a probabilistic polynomial-time *stateful* simulator $\mathsf{Sim}$, such that no probabilistic polynomial-time adversary can distinguish whether it is in $\mathsf{Expt}^{\mathsf{real}}$ or $\mathsf{Expt}^{\mathsf{sim}}$ except with negligible probability, i.e., Specifically, in $\mathsf{Expt}^{\mathsf{real}}$, the adversary is interacting with the real-world algorithms, whereas in $\mathsf{Expt}^{\mathsf{sim}}$ the adversary is interacting with a simulator $\mathsf{Sim}$ that sees only $\mathsf{tr}(\mathsf{bp}, x)$, but not the $\mathsf{bp}$ or $x$ themselves.




**Garbling layered branching programs: construction.** Figure 7 shows a construction for garbling layered branching programs. The idea is that each non-sink node contains a tuple

$$\left(v.\mathsf{addr},\ v.\mathsf{payload},\ \boxed{\mathsf{E}_{\mathsf{L}_0}(\mathsf{sk}_{v0}),\ \mathsf{E}_{\mathsf{L}_1}(\mathsf{sk}_{v1})}_{\ \mathsf{permute}}\right)$$

where $v.\mathsf{addr}$ denotes the address in the input this node looks at, $v.\mathsf{payload}$ denotes the payload of this node, and $\mathsf{E}_{\mathsf{L}_0}(\mathsf{sk}_{v0}),\ \mathsf{E}_{\mathsf{L}_1}(\mathsf{sk}_{v1})$ are two ciphertexts that can be decrypted with the a corresponding garbled label for input bit $x_i$. Depending on whether $x_i = 0$ or $x_i = 1$, its garbled label can decrypt exactly one of these ciphertexts, to obtain $\mathsf{sk}_{v0}$ or $\mathsf{sk}_{v1}$. Then, $\mathsf{sk}_{v_b}$ will inductively be used to encrypt the $b$-th child node of $v$ for $b \in \{0, 1\}$. For security, it is important that the two ciphertexts be permuted in random order.

Given this garbling scheme, evaluation is defined in the obvious manner: given a secret key $\mathsf{sk}_v$ to a node $v$, the evaluator first decrypts the current node, which reveals an input bit to look at. The evaluator grabs the corresponding garbled label for the input bit, decrypts one of the ciphertexts, and obtains a secret key for decrypting one node in the next layer. We stress that the evaluator can decrypt only the nodes on the evaluation path, and thus learns these nodes' payload and input addresses. The evaluator cannot decrypt any non-evaluation-path nodes, since the evaluator knows only one garbled label for each input bit.

**Optimizations.** We can apply standard optimizations described by Bellare et al. [5] such that the decryptor need not use trial-and-error during decryption, but will know exactly which child node to decrypt. Another optimization they describe [5] avoids the ciphertext having to carrying the $\mathsf{nonce}$ term which causes ciphertext expansion. To keep our notations simple, we did not include these optimizations in our basic exposition, but we suggest that they be applied in a practical implementation.

**Theorem 2.** *Assume that* $\mathsf{PRF}$ *is a secure pseudorandom function. Then, the garbling scheme for layered branching programs as described in Figure 7 satisfies both correctness and security requirements.*

*Proof.* We construct the simulator in Figure 9.

**Garbling Scheme for Layered Branching Programs**

- $\mathsf{KeyGen}(1^\lambda)$: Output a random $\mathsf{sk} \leftarrow \{0,1\}^\lambda$.
- $\mathsf{GarbleData}(\mathsf{sk}, x)$: For each input bit $x_i$, compute $\overline{x}_i := \mathsf{PRF}(\mathsf{sk}, i||x_i)$. Output $\overline{x} := (\overline{x}_1, \ldots, \overline{x}_m)$.
- $\mathsf{GarbleProg}(\mathsf{sk}, \mathsf{bp})$: Let $S$ denote the source node. Call $\mathsf{GarbleNode}(\mathsf{sk}, S)$, and output

$$(\mathsf{sk}_S,\ \{\overline{v} : \forall v \in \mathsf{bp}\})$$

where $\mathsf{sk}_S$ and $\overline{v}$ are generated and saved during recursive call to $\mathsf{GarbleNode}(\mathsf{sk}, S)$. Further, the garbled nodes are output layer by layer, where nodes in each layer are output in a **random order**.

- $\mathsf{Eval}(\overline{\mathsf{bp}}, \overline{x})$: Output $\mathsf{EvalOnce}(1, \mathsf{sk}_S, \overline{\mathsf{bp}}, \overline{x})$ — see Figure 8 for the definition of $\mathsf{EvalOnce}$.

$\mathsf{GarbleNode}(\mathsf{sk}, v)$

- If node $v$ has been garbled before, just return the same answer as before.
- If $v$ is a sink node: pick a random key $\mathsf{sk}_v \in \{0,1\}^\lambda$, and return $(\mathsf{sk}_v, \overline{v})$ where the garbled node $\overline{v}$ is defined as follows:

$$\overline{v} := \mathsf{E}_{\mathsf{sk}_v}(v.\mathsf{payload})$$

- If $v$ is not a sink node, let $v_0$ denote the child of $v$ following the 0 edge, and let $v_1$ denote the child of $v$ following the 1 edge.
  - Call $(\mathsf{sk}_{v_0}, \overline{v}_0) := \mathsf{GarbleNode}(v_0)$ and $(\mathsf{sk}_{v_1}, \overline{v}_1) := \mathsf{GarbleNode}(v_1)$. In other words, recursively call $\mathsf{GarbleNode}$ to garble both of $v$'s children.
  - Recall that node $v$ looks at input bit $x[v.\mathsf{addr}]$[a]. Let $\mathsf{L}_0$ denote the garbled label for $x[v.\mathsf{addr}] = 0$, and $\mathsf{L}_1$ denote the garbled label for $x[v.\mathsf{addr}] = 1$. More formally, let $\mathsf{L}_b := \mathsf{PRF}(\mathsf{sk}, v.\mathsf{addr}||b)$ for $b \in \{0,1\}$. Pick a random encryption key $\mathsf{sk}_v \in \{0,1\}^\lambda$, and compute

$$\overline{v} := \mathsf{E}_{\mathsf{sk}_v}\left(v.\mathsf{addr},\ v.\mathsf{payload},\ \boxed{\mathsf{E}_{\mathsf{L}_0}(\mathsf{sk}_{v_0}),\ \mathsf{E}_{\mathsf{L}_1}(\mathsf{sk}_{v_1})}_{\,\mathsf{permute}}\right)$$

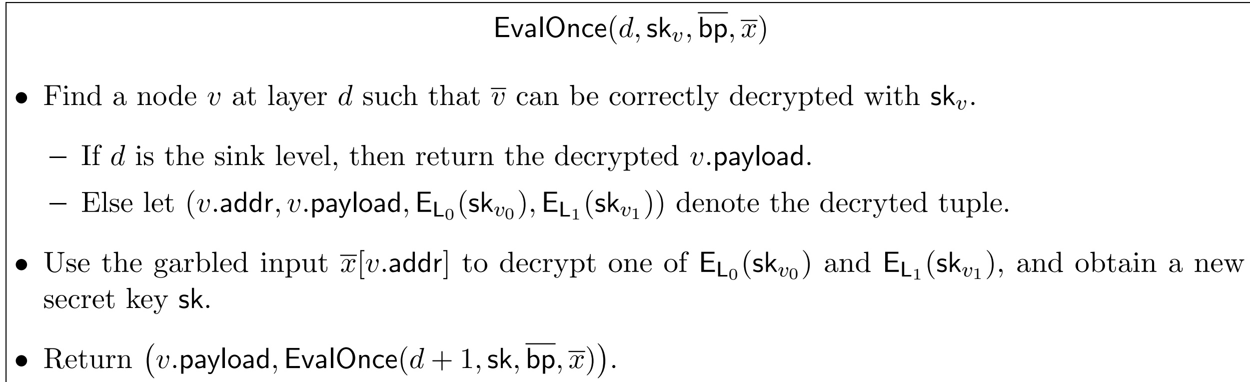
    where $v.\mathsf{payload}$ is also leaked by $\mathsf{tr}(\mathsf{bp}, x)$, and $\boxed{\phantom{xx}}_{\mathsf{permute}}$ denotes two terms in randomly permuted order.
  - Return $(\mathsf{sk}_v, \overline{v})$.

[a] $x[v.\mathsf{addr}]$ denotes the input bit $x_i$ where $i = v.\mathsf{addr}$

$\mathsf{E}_{\mathsf{L}}(m)$

- Let $\mathsf{PRF} : \{0,1\}^{|\mathsf{L}|} \times \{0,1\}^\lambda \to \{0,1\}^{|m|}$ denote a pseudorandom function.
- Output $(\mathsf{PRF}(\mathsf{L}, \mathsf{nonce}) \oplus m,\ \mathsf{nonce})$ where $\mathsf{nonce} \in \{0,1\}^\lambda$ is a freshly generated nonce.

**Figure 7:** Garbling scheme for layered branching programs, with subroutine $\mathsf{GarbleNode}$ recursively defined.



$\mathsf{EvalOnce}(d, \mathsf{sk}_v, \overline{\mathsf{bp}}, \overline{x})$

- Find a node $v$ at layer $d$ such that $\overline{v}$ can be correctly decrypted with $\mathsf{sk}_v$.
  - If $d$ is the sink level, then return the decrypted $v.\mathsf{payload}$.
  - Else let $(v.\mathsf{addr}, v.\mathsf{payload}, \mathsf{E}_{\mathsf{L}_0}(\mathsf{sk}_{v_0}), \mathsf{E}_{\mathsf{L}_1}(\mathsf{sk}_{v_1}))$ denote the decryted tuple.
- Use the garbled input $\overline{x}[v.\mathsf{addr}]$ to decrypt one of $\mathsf{E}_{\mathsf{L}_0}(\mathsf{sk}_{v_0})$ and $\mathsf{E}_{\mathsf{L}_1}(\mathsf{sk}_{v_1})$, and obtain a new secret key $\mathsf{sk}$.
- Return $\big(v.\mathsf{payload}, \mathsf{EvalOnce}(d+1, \mathsf{sk}, \overline{\mathsf{bp}}, \overline{x})\big)$.

**Figure 8:** Evaluate the garbled branching program at one level, recursively defined.

**Simulator for garbling input.** When the simulator is asked to garble an input, the simulator cannot see the input. The simulator simply picks random strings from $\overline{x}_i := \{0,1\}^\lambda$ and returns them. The simulator also remembers $\overline{x}_i$.

**Simulator for garbling branching program.** When the simulator is asked to garble a branching program, the simulator is only given $\mathsf{tr}(\mathsf{bp}, x)$, but not $\mathsf{bp}$ or $x$. The simulator picks a random evaluation path containing a random node from each level. For nodes on the evaluation path, the simulator garbles them by making a recursive call to the following $\mathsf{SimGarbleNode}(\mathsf{S})$ where $S$ denotes the source:

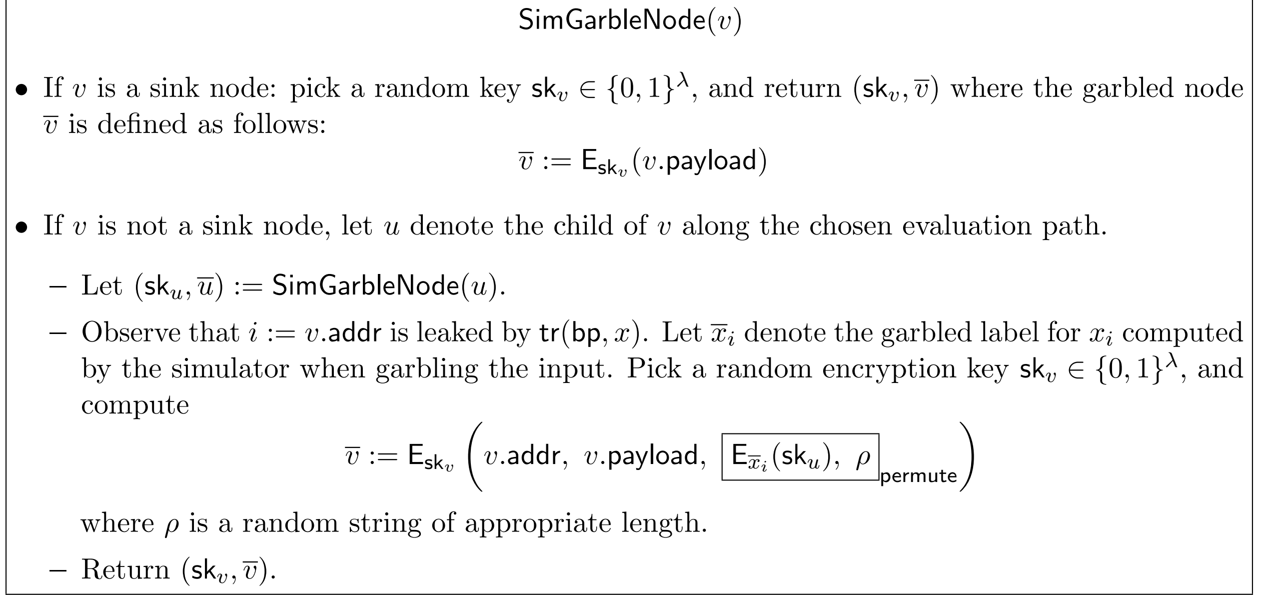
To garble all other nodes in the $\mathsf{bp}$, the simulator outputs random strings of appropriate length.

**Indistinghishability of the two experiments.** Indistinguishability of the two experiments is not hard to show by defining a sequence of hybrid games. The proof technique is standard and is similar to that of Bellare et al. [5]. First, the garbled inputs computed from pseudorandom functions are now replaced with true randomness as in the simulation, assuming that the pseudorandom function $\mathsf{PRF}$ is secure. Next, for all encryptions under garbled labels that are not given out as part of the garbled input, replace these ciphertexts with random. Next, the ciphertexts corresponding to non-excution paths in the garbled branching program are replaced one by one from real to random, going from the source level to the sink level. Adjacent hybrid games are indistinguishable following the security of the PRF function. □

### 4.3 Extension: Temporal Garbling of Layered Branching Programs

Later in our application of garbled branching programs, the inputs of the branching program are Bloom filter bits, and the branching program itself depends on the logical index of memory to be accessed. It is easy to assign all Bloom filter bits a globally unique address (unique over the lifetime of the ORAM). However, we observe that both the inputs (i.e., Bloom filter bits) and the branching programs themselves are generated adaptively over time. We stress that at any time, the branching program to be queried depends only on the input bits that have already been generated — this lets us get around an adaptive security issue easily.

Therefore, we extend our basic garbled branching program to a temporal version to suit the needs of the application. A *temporal* garbling scheme for the family $\mathcal{BP}[m, n_1, n_2, \ldots, n_D]$ contains the following algorithms:



$\mathsf{SimGarbleNode}(v)$

- If $v$ is a sink node: pick a random key $\mathsf{sk}_v \in \{0,1\}^\lambda$, and return $(\mathsf{sk}_v, \overline{v})$ where the garbled node $\overline{v}$ is defined as follows:
$$\overline{v} := \mathsf{E}_{\mathsf{sk}_v}(v.\mathsf{payload})$$
- If $v$ is not a sink node, let $u$ denote the child of $v$ along the chosen evaluation path.
  - Let $(\mathsf{sk}_u, \overline{u}) := \mathsf{SimGarbleNode}(u)$.
  - Observe that $i := v.\mathsf{addr}$ is leaked by $\mathsf{tr}(\mathsf{bp}, x)$. Let $\overline{x}_i$ denote the garbled label for $x_i$ computed by the simulator when garbling the input. Pick a random encryption key $\mathsf{sk}_v \in \{0,1\}^\lambda$, and compute
$$\overline{v} := \mathsf{E}_{\mathsf{sk}_v}\left(v.\mathsf{addr},\ v.\mathsf{payload},\ \boxed{\mathsf{E}_{\overline{x}_i}(\mathsf{sk}_u),\ \rho}_{\mathsf{permute}}\right)$$
    where $\rho$ is a random string of appropriate length.
  - Return $(\mathsf{sk}_v, \overline{v})$.

**Figure 9:** A simulator for garbling a node on the simulated evaluation path.

.

- $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$: On input $1^\lambda$, generates a secret key $\mathsf{sk}$.
- $\overline{x}_i \leftarrow \mathsf{GarbleData}(\mathsf{sk}, i, x)$: Given secret key $\mathsf{sk}$ and the $i$-th bit of input $x_i$, outputs the garbled input bit $\overline{x}_i$.
- $\overline{\mathsf{bp}} \leftarrow \mathsf{GarbleProg}(\mathsf{sk}, \mathsf{bp})$: Given a secret key $\mathsf{sk}$ and a layered branching program $\mathsf{bp} \in \mathcal{BP}[m, n_1, \ldots, n_D]$, outputs a garbled branching program $\overline{\mathsf{bp}}$.
- $\mathsf{res} \leftarrow \mathsf{Eval}(\overline{\mathsf{bp}}, \overline{x})$: Given garbled $\overline{\mathsf{bp}}$, and garbled input $\overline{x}$, evaluates the outcome $\mathsf{res}$.

Correctness is defined in a similar way as before. We modify the security definition as follows, where we allow the adversary to adaptively specify input bits and programs to garble, however, with the following restrictions:

- Every branching program specified by the adversary must depend only on the inputs submitted earlier by the adversary. Specifically, we use the notation $x_{|t}$ to denote the input string restricted to the addresses queried thus far by the adversary.
- The adversary must not query both bits for the same input address $i$.

If a probablistic polynomial-time adversary satisfies the above constraints, we say that it is a *compliant* adversary.

**Figure 10:** Block request algorithm as a layered branching program.

| $\mathsf{Expt}^{\mathsf{real}}$ | $\mathsf{Expt}^{\mathsf{sim}}$ |
|---|---|
| $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$, $\overline{\mathsf{bp}} := \bot$ | $\overline{\mathsf{bp}} = \bot$ |
| For $t = 1, 2, \ldots, \mathsf{poly}(\lambda)$: | For $t = 1, 2, \ldots, \mathsf{poly}(\lambda)$: |
| $(i, x_i) \leftarrow \mathcal{A}(1^\lambda, \overline{\mathsf{bp}})$ | $(i, x_i) \leftarrow \mathcal{A}(1^\lambda, \overline{\mathsf{bp}})$ |
| $\overline{x}_i \leftarrow \mathsf{GarbleData}(\mathsf{sk}, i, x_i)$ | $\overline{x}_i \leftarrow \mathsf{Sim}(1^\lambda)$ |
| $\mathsf{bp} \leftarrow \mathcal{A}(\overline{x}_i)$ | $\mathsf{bp} \leftarrow \mathcal{A}(\overline{x}_i)$ |
| $\overline{\mathsf{bp}} \leftarrow \mathsf{GarbleProg}(\mathsf{sk}, \mathsf{bp})$ | $\overline{\mathsf{bp}} \leftarrow \mathsf{Sim}(\mathsf{tr}(\mathsf{bp}, x_{\vert t}))$ |
| $b \leftarrow \mathcal{A}(\overline{\mathsf{bp}})$ | $b \leftarrow \mathcal{A}(\overline{\mathsf{bp}})$ |

We say that a temporal garbling scheme for layered branching programs is secure, if there exists a probabilistic polynomial-time simulator $\mathsf{Sim}$, such that no *compliant* probabilistic polynomial-time adversary can distinguish $\mathsf{Expt}^{\mathsf{real}}$ and $\mathsf{Expt}^{\mathsf{sim}}$ except with negligible probability.

**Construction and proof.** Construction and proof of the temporal version follow in a straightforward manner from our basic construction described in Section 4.2. Because our security game requires that every branching program submitted by the adversary depend only on input bits that have been queried before, there is no adaptive security issue, and the proof of Theorem 2 can be extended trivially for the temporal version.

## 4.4 Block Request as a Layered Branching Program

We now describe why the block request algorithm (online phase) can be modeled as a layered branching program.

**Theorem 3.** *The block access algorithm (online phase) as described in Figure 4 is a layered branching program with $O(1)$ width, and $O(kL)$ depth, where $k$ is the number of hashes in the Bloom filter, and $L$ is the number of levels in the ORAM hierarchy.*

*Proof.* To show this, we describe what the inputs to the branching program are, and how the branching program is constructed.

**Inputs of the layered branching program.** The inputs of this branching program are Bloom filter bits. For the entire lifetime of the ORAM, we assign a globally unique address to every Bloom filter bit:

$$[\text{global address of BF bit}] := (T,\ \mathsf{BFOffset})$$

where $T$ denotes the time at which the level is rebuilt (note that $T$ implies the level number), and BFOffset is the offset of the bit within the Bloom filter.

**Branching program.** Our "block request" branching program is parameterized by the logical index denoted idx of the block to be requested, as well as the current time $T$. Given idx and $T$, the branching program is fixed, and can be constructed as in Figure 10. Each blue or red rectangle is a super node that denotes the behavior of the branching program at one level in the ORAM hierarchy. If a block has been found in a level, then the branching program will permanently transition into red supernodes, in which the branching program will look at $k$ random addresses in the Bloom filter. Inside each blue super-node, however, the branching program looks at $k$ real locations in the Bloom filter where the locations are pre-computable given the block idx to be requested, and the current time $T$. Each blue or red supernode looks at $k$ locations in the Bloom filter one by one, thus each supernode has $k$ internal layers to keep track of whether a Bloom filter bit of 0 has been encountered. At the end of the $k$ layers inside a supernode, we enter a state which either discloses the real hkey or a mask hkey, both of which can be precomputed knowing the block idx and the current time $T$.

For convenience, Figure 10 introduceds a special type of edge called a shortcut edge. A shortcut edge does not read any input bits, but simply shortcuts into the next node. Shortcut edges can be easily supported if we add some fake bits into the input stream, and the shortcut edges basically transition into the next node regardless of whether the fake input bits are 0 or 1.

□

### 4.5 Single Online Roundtrip with Garbled Branching Programs

**Garbling transformation.** It suffices to make the following modifications to the ORAM algorithm described in Section 3.

- When performing oblivious rebuild of a level's Bloom filter (see Figure 6), instead of using encrypted bits for each Bloom filter bit, we replace each encrypted bit with its corresponding garbled label.
- For the online phase of block request algorithm (Figure 4), instead of the client evaluating the algorithm of Figure 4 fetching data from the server as necessary, the client now treats the program in Figure 4 as a layered branching program, garbles it, and sends the garbled version to the server.
- The server now evaluates the garbled branching program representing the client's query, and obtains exactly one hkey for each level of the ORAM hierarchy.

**Corollary 1** (Garbling transformation preserves security)**.** *Assume that the temporal garbling scheme for layered branching programs satisfies simulation security as described in Section 4.3. If the underlying ORAM scheme described in Section 3 achieves malicious security by Definition 2 (see Appendix B), then the transformed scheme as described above also achieves malicious security by Definition 2.*

*Proof.* (sketch.) Let $\mathcal{A}$ denote any probabilistic polynomial-time adversary for the transformed, single-round scheme. We now consider the following adversary $\mathcal{A}'$ for the underlying ORAM scheme. Let Sim denote the simulator for the garbling scheme. Roughly speaking, $\mathcal{A}'$ receives protocol messages (consisting of physical addresses or hkeys to fetch), and the $\mathcal{A}'$ runs Sim whose output,

containing garbled Bloom filter bits and garbled programs, will be relayed to $\mathcal{A}$ whenever appropriate (in place of the encrypted Bloom filter bits and trace of the branching program evaluation). Now whatever $\mathcal{A}$ outputs, $\mathcal{A}'$ simply relays it back to the honest client (or the ideal-world simulator $\mathcal{S}$ during a simulation). Now if $\mathcal{S}$ corresponds to an ideal-world simulator for $\mathcal{A}'$ for the underlying ORAM scheme, it is not hard to see that the union of $\mathcal{S}$ and the garbling scheme's $\mathsf{Sim}$ would be an ideal-world simulator for $\mathcal{A}$ for the transformed, single-round scheme. □

### 4.6 Cost Analysis

**Offline cost.** The offline cost is almost the same as in Section 3, except that now, in the last step of the Bloom filter creation, the client sequentially writes garbled labels for each Bloom filter bit, incurring an extra $O(\lambda^2 n)$ bits in bandwidth cost, where $n$ is the size of the level being rebuilt, and $\lambda$ decides the security failure $2^{-O(\lambda)}$. For simplicity, here we assume that a garbled label of $O(\lambda)$ bits has a security failure probability of $2^{-O(\lambda)}$ for the garbling scheme.

**Online cost.** The client still retrieves $O(\log N)$ blocks in the online phase. We now analyze the cost of metadata for the online request phase. The metadata is for sending a garbled branching program of $O(1)$ width and $kL$ depth. Each garbled node incurs $O(\lambda)$ bits in length. For simplicity, here we assume that the $\mathsf{PRF}$ and the encryption schemes have $O(\lambda)$-bit inputs and outputs, and attain a security failure of $2^{-O(\lambda)}$. The total length of this garbled branching program is thus $O(\lambda k L) = O(\lambda^2 \log N)$.

**Block size for attaining $O(\log N)$ bandwidth blowup.** To achieve a security failure negligible in $N$, we let $\lambda := O(\log N)\omega(1)$. In this case, the extra online/offline costs mentioned above are absorbed asymptotically in comparison with the cost of Bloom filter rebuilding.

Therefore, if we set the block size to be $\omega(\log^3 N)$, the total cost of metadata gets absorbed in the asymptotics, and we obtain $O(\log N)$ bandwidth blowup.

## 5 Constant Bandwidth Blowup, Single Online Roundtrip ORAM

We show how to extend our construction to achieve constant bandwidth blowup borrowing techniques from Onion ORAM by Devadas et al. [12].

The basic idea of Onion ORAM is to have the server select blocks in a PIR (Private Information Retrieval) fashion to save bandwidth on both requests and eviction. On a request, the client uses PIR to retrieve the block of interest instead of downloading $O(\log N)$ blocks. On evictions, the operation is similar to PIR, except that the server simply moves the output block to the eviction destination (the rebuilt level for Bucket ORAM), and does not need to return anything to the client. Every such eviction operation will add an additional layer of encryption to the blocks involved. Onion ORAM proposed a new eviction strategy that ensures "steady progress" (i.e., blocks do not get stuck during evictions), and showed that with such an eviction strategy, the number of encryption layers can be bounded.

Interestingly, Bucket ORAM's level rebuild also has the "steady progress" property since blocks always move to the newly rebuilt layer and never get stuck. Therefore, we can directly adopt Onion ORAM's eviction techniques to get constant bandwidth blowup on evictions. The only challenge is on requests: a standard PIR protocol requires multiple roundtrips. We will address this problem by including the PIR help data in block metadata. Below we explain Onion ORAM's techniques and how to adopt them to Bucket ORAM.

### 5.1 Building Block: PIR using Additively Homomorphic Encryption

In the PIR setting, a server holds $m$ data blocks $\mathsf{block}_1, \mathsf{block}_2, \cdots, \mathsf{block}_m$, a client wishes to download $\mathsf{block}_{i^*}$ without revealing $i^*$. One simple way to implement PIR is to use an additively homomorphic encryption (AHE) scheme $E(\cdot)$. The client sends an encrypted select vector $\{E(x_1), E(x_2), \cdots, E(x_m)\}$ where $x_i = 1$ iff $i = i^*$ and $x_i = 0$ otherwise. The server breaks up each block into multiple chunks $\mathsf{block}_i = \{\mathsf{block}_i[1], \mathsf{block}_i[2], \cdots\}$ where each chunk is in the plaintext space of $E(\cdot)$. Then for each chunk, the server evaluates $\bigoplus_i (E(x_i) \otimes \mathsf{block}_i[j]) = E\left(\sum_i (x_i \cdot \mathsf{block}_i[j])\right) = E(\mathsf{block}_{i^*}[j])$. The output is the encrypted $j$-th chunk of the requested $\mathsf{block}_{i^*}$. The server just aggregates all the chunks and returns the requested block to the client. When the block size is sufficiently large and dominates all the encrypted select bits, the bandwidth blowup is $O(1)$.

Note that we assumed plaintext input blocks above, and the ouput is the encrypted block of interest. If the input blocks are already encrypted, then the output will acquire an additional layer of encrpytion. In fact, we will see below that in Onion ORAM and Bucket ORAM, blocks become onion-encrypted as the above PIR operation is performed repeatedly.

### 5.2 Constant Bandwidth Blowup for Eviction/Level Rebuild

Like Onion ORAM, we require $L + 1$ additively homomorphic encryption schemes $\{E_0(\cdot), E_1(\cdot), \cdots, E_L(\cdot)\}$ where the ciphertext space of $E_{\ell-1}$ is in the plaintext space of $E_\ell$ for all $1 \leq \ell \leq L$. Then a block can be onion-encrypted using these encryption schemes in increasing order, i.e., $E_\ell(E_{\ell-1}(\ldots E_0(\cdot)))$. We will use $E^\ell(\cdot)$ as a shorthand notation for the above layered encryption.

Assume blocks at level $\ell$ have $\ell$ layers of encryption (will be shown using induction). When rebuilding level $\ell + 1 < L$, each slot in level $\ell + 1$ will be taken by one of the $2Z = O(\log N)\omega(1)$ blocks at level $\ell$ ($Z$ from the real level $\ell$ and $Z$ from the transient level $\ell$). Denote them $E^\ell(\mathsf{block}_1), E^\ell(\mathsf{block}_2), \cdots, E^\ell(\mathsf{block}_{2Z})$. A PIR-like protocol can obliviously move any $i^*$-th candidate block into the slot without revealing $i^*$. The client sends encrypted select bits $E_{\ell+1}(x_i)$ (not onion encrypted) to the server, where each $x_i$ is encoded as described above. The server evaluates for each chunk $j$, $\bigoplus_i \left(E_{\ell+1}(x_i) \otimes E^\ell(\mathsf{block}_i[j])\right)$, which yields $E_{\ell+1}(E^\ell(\mathsf{block}_{i^*}[j])) = E^{\ell+1}(\mathsf{block}_{i^*}[j])$. This shows that blocks at level $\ell + 1$ have $\ell + 1$ layers of encryption and completes the induction.

Rebuilding the last level (level $L - 1$) requires additional care. Unlike all the other levels, level $L - 1$ is not empty before rebuilding, and some blocks may remain in the rebuilt last level (violating "steady progress"). If we simply include blocks in level $L - 1$ as the input of PIR, the number of encryption layers at level $L - 1$ will increase by one after each level rebuild and grow unbounded. To get around this issue, we need the client to preprocess each slot in level $L - 1$ before rebuilding it (similar to leaf postprocessing in Onion ORAM). The preprocessing step simply requires the client to download the slot, peel off 2 layers and reencrypt with $E^{L-2}$, and send it back to the server. The $E^{L-2}$ ciphertexts can be safely included as input to the PIR protocol. The bandwidth blowup introduced by the preprocessing is amortized $O(1)$, since level $L - 1$ is rebuilt only every $\Theta(2^{L-1}Z')$ accesses, on the same order of the total number of slots in the level.

### 5.3 Constant Bandwidth Blowup for Requests

Block requests can be posed as a PIR problem. The server is holding $\Theta(\log N)$ candidate blocks (one per level); the client only wants one of them, but does not want to reveal which one. Blocks

from different levels will have different number of layers, but the server can "lift" all blocks to $E^{L-1}(\cdot)$ by repeatedly encrypting them.

The remaining challenge for Bucket ORAM is how the server gets the encrypted select bits. In Onion ORAM, the client just downloads the metadata for all $\Theta(\log N)$ blocks to figure out which block she wants, and can then easily construct the encrypted select bits and send to the server. But this would require multiple roundtrips.

Instead, on every level rebuild, we introduce a new field in the metadata for each block: an AHE encrypted valid bit $E_L(\mathsf{isvalid})$. Real blocks have $\mathsf{isvalid} = 1$, mask blocks have $\mathsf{isvalid} = 0$, and dummy blocks have "don't care" values for this field — since dummy blocks will not be fetched during the online phase. The reason we use $E_L(\cdot)$ is that all the lifted blocks live in the ciphertext space of $E^{L-1}(\cdot)$. Recall that out of the candidate blocks, only the block of interest is valid (has $E_L(1)$), all the other blocks are mask blocks (have $E_L(0)$), so these encrypted valid bits can be directly used in the PIR protocol. Since a real block will be read only once before the level is rebuilt, there is no need to update $E_L(\mathsf{isvalid})$ other than during level rebuild.

### 5.4 Encryption Details

For a block at level $\ell$, its payload is homomorphically encrypted in $\ell$ onion layers. To get constant bandwidth, we require a constant ciphertext expansion after up to $L$ layers of encryption. As noted in the Onion ORAM paper [12], the Damgård-Jurik cryptosystem [9] has an additive ciphertext expansion per layer and satisfies this property. For every block (at whichever level), its valid bit is homomorphically encrypted using $E_L$ (not onion-encrypted), and the other metadata is encrypted simply using a symmetric-key encryption like AES. Specifically, let $E$ be the Damgård-Jurik encryption scheme,

$$\mathsf{block} := \mathsf{data}||\mathsf{metadata} \quad \overset{\text{encrypt}}{\Longrightarrow} \quad \Big(E^{\ell}(\mathsf{data}), E_L(\mathsf{isvalid}), \mathsf{AES}(\mathsf{other\ metadata})\Big)$$

In the above, we omit writing the encryption keys for simplicity.

Other auxiliary data, such as the server-side garbled bloom filter will be constructed in the same way as in Section 4

### 5.5 Putting it Together and Malicious Security

**Theorem 4.** *Assuming pseudorandom functions exist and Decisional Composite Residuosity (DCR) is hard, the construction in this section is a server-computation ORAM with single online roundtrip, $O(1)$ bandwidth blowup, $O(1)$ blocks of client storage, $O(N)$ blocks of server storage, when the block size is $\widetilde{\Omega}(\log^5 N)$. Assuming the existence of collision-resistant hash functions, the construction can be extended to a malicious server with the same performance when the block size is $\widetilde{\Omega}(\log^6 N)$.*

We refer the readers to [12] for a detailed analysis on block size requirements and how to achieve security under a malicious server.

## Acknowledgments

This work is funded in part by NSF grants CNS-1314857, CNS-1453634, CNS-1518765, CNS-1514261, a Packard Fellowship, a Sloan Fellowship, two Google Faculty Research Awards, and a VMWare Research Award.

Emil Stefanov was supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0946797, and by the DoD National Defense Science and Engineering Graduate Fellowship.

Muhammad Naveed is supported in part by the Google PhD Fellowship and Sohaib and Sara Abbasi Fellowship.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

## References

[1] https://en.wikipedia.org/wiki/Bloom_filter.

[2] ObliVM. www.oblivm.com.

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

[4] D. Asonov and J.-C. Freytag. Almost optimal private information retrieval. In *PET*, 2003.

[5] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[6] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[7] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[8] K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. In *Asiacrypt*, 2014.

[9] 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.

[10] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.

[11] 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, Aug. 2014.

[12] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *TCC*, 2016.

[13] 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*, pages 3–8, Oct. 2012.

[14] C. W. 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 *HPCA*, pages 213–224, 2014.

[15] S. Garg, S. Lu, R. Ostrovsky, and A. Scafuro. Garbled RAM from one-way functions. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 449–458, 2015.

[16] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled RAM revisited. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 405–422, 2014.

[17] C. Gentry, S. Halevi, M. Raykova, and D. Wichs. Outsourcing private RAM computation. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 404–413, 2014.

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

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

[20] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[21] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 95–100, New York, NY, USA, 2011. ACM.

[22] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[23] A. Iliev and S. W. Smith. Protecting client privacy with trusted computing at the server. *IEEE Security and Privacy*, 3(2):20–28, Mar. 2005.

[24] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[25] C. Liu, M. Hicks, A. Harris, M. Tiwari, M. Maas, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[26] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating Efficient RAM-model Secure Computation. In *S & P*, May 2014.

[27] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376, 2015.

[28] S. Lu and R. Ostrovsky. How to garble RAM programs. In *Eurocrypt*, 2013.

[29] 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.

[30] T. Moataz, T. Mayberry, and E. Blass. Constant communication ORAM with small blocksize. In *ACM CCS*, 2015.

[31] T. Moataz, T. Mayberry, and E.-O. Blass. Constant communication oram with small blocksize. Cryptology ePrint Archive, Report 2015/570, 2015.

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

[33] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[34] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[35] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious RAM. In *Usenix Security*, 2015.

[36] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

[37] 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.

[38] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Syst. J.*, 40(3):683–695, Mar. 2001.

[39] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy*, 2013.

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

[41] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[42] X. S. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS*, 2015.

[43] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[44] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.

[45] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.

[46] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.

# A Notations

Table 2 is a notation table containing the global variables adopted in this paper.

**Table 2:** Notations

| Variable | Meaning |
|---|---|
| $\lambda$ | security parameter to obtain $2^{-O(\lambda)}$ a security failure probability |
| $N$ | total number of real blocks |
| $L = O(\log N)$ | number of levels in the ORAM hierararchy |
| $B$ | number of bits in each block |
| $T$ | time a block is last written, a level was last rebuilt, or the current time |
| $\mathsf{label}$ | leaf label for a block |
| $k = O(\lambda)$ | number of hashes for the bloom filter |
| $Z = O(\log N)\omega(1)$ | bucket size |
| $Z' = Z/O(1)$ | size of the client's local eviction buffer |

# B Definition of Server-Computation ORAM

We directly adopt the definitions and notations used by Apon et al. [3] 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 [7]. 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 $\beta$; and the server's input is $\perp$. At the end of the $\mathsf{Setup}$ 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 $\mathsf{op}$ where $\mathsf{op} := (\mathsf{read}, \mathsf{addr})$ or $\mathsf{op} := (\mathsf{write}, \mathsf{addr}, \mathsf{data})$; the server starts in state $Z$, and has no input. In a correct execution of the protocol, the client's output $\mathsf{data}$ is the current value of the memory $D$ at location $\mathsf{addr}$ (for writes, the output is the old value of $D[\mathsf{addr}]$ 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 $m = \text{poly}(\lambda)$, an $\mathsf{op} := (\mathsf{read}, \mathsf{addr})$ operation would always return the last value written to the logical location $\mathsf{addr}$ (except with negligible probability).

## B.1 Security Definition

We adopt a standard simulation-based definition of secure computation [7], 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 $\mathsf{ok}$ or $\mathsf{abort}$ to $\mathcal{F}$. $\mathcal{F}$ then says $\mathsf{ok}$ or $\perp$ to the client accordingly.

- *Access.* In each time step, the environment $\mathcal{Z}$ specifies an operation $\mathsf{op} := (\mathsf{read}, \mathsf{addr})$ or $\mathsf{op} := (\mathsf{write}, \mathsf{addr}, \mathsf{data})$ as the client's input. The client sends $\mathsf{op}$ to $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ (without revealing to $\mathcal{S}$ the operation $\mathsf{op}$). If $\mathcal{S}$ says $\mathsf{ok}$ to $\mathcal{F}$, $\mathcal{F}$ sends $D[\mathsf{addr}]$ to the client, and updates $D[\mathsf{addr}] := \mathsf{data}$ accordingly if this is a write operation. The client then forwards $D[\mathsf{addr}]$ to the environment $\mathcal{Z}$. If $\mathcal{S}$ says $\mathsf{abort}$ 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 $\mathsf{Setup}$ protocol with the server $\mathcal{A}$. Then at each time step, $\mathcal{Z}$ specifies an input $\mathsf{op} := (\mathsf{read}, \mathsf{addr})$ or $\mathsf{op} := (\mathsf{write}, \mathsf{addr}, \mathsf{data})$ to the client. The client then runs the $\mathsf{Access}$ 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* $\mathsf{negl}$ *such that*

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

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.

# C Malicious Security for the Scheme in Section 3

We can obtain malicious security with standard techniques. Most of our data and metadata satisfy *predictive time*, i.e., the client can efficiently compute the time at which this block (or metadata) was last written to the server during a level rebuild.

With the exception of some metadata, we can, for the most part, use time- and location-aware message authentication codes to achieve malicious security (and there is no need for building a Merkle-hash tree). Upon retrieving a block from the server, the client always verifies the message authentication code, and rejects block if the verification fails.

In our ORAM scheme in this section, block data can be accessed in two modes:

1. By their $\mathsf{hkey}$s during the online phase of the block access algorithm;
2. By their explicit physical addresses on the server (typically containing the level number, the bucket number, and the offset within the bucket) during the offline shuffling phase.

All auxiliary metadata (including the per-level bloom filters and transient metadata created during level rebuilding, not including metadata attached to blocks) are always accessed by their explicit physical addresses.

Therefore, below we discuss how to authenticate auxiliary metadata and block data separately.

**Authenticating auxiliary metadata.** Observe that in our scheme described in this section, all *auxiliary* metadata (including the per-level bloom filters and transient metadata created during level rebuilding, not including metadata attached to blocks) are accessed by their explict addresses. Further, all metadata touched during level rebuilding is written to the server via linear scans or oblivious sorting. Both of these operations perform each write to each physical location at public and pre-determined times.

Putting the above observations together, the client can attach a time- and location-sensitive message authentication code $\mathsf{MAC}(\mathsf{sk}, T||\mathsf{phys_addr}||\mathsf{metadata})$ to every auxiliary metadata chunk, where $T$ is the time $\mathsf{metadata}$ was last written to the server.

**Authenticating block data.** Block data (including most directly attached to the block, such as $\mathsf{idx}$ the leaf $\mathsf{label}$) can be accessed by either their $\mathsf{hkey}$ or their physical address on the server. Block data are authenticated also using a time- and location-aware MAC:

$$\mathsf{MAC}(\mathsf{sk}, T||\mathsf{phys_addr}||\mathsf{block})$$

where $T$ denotes the last time the block was written, and $\mathsf{phys_addr}$ typically contains the level number, bucket number, and offset within the bucket.

The only subtlety is that when the block is accessed by its $\mathsf{hkey}$, the server needs to additionally return the block's $\mathsf{phys_addr}$ to the client, such that the client is able to verify the MAC.

**Subtlety.** The only exception to this "predictive time" rule is the client invaldating blocks fetched in the online phase, by setting $\mathsf{block.type} :=$ "dummy". Therefore, the client can employ a merkle hash tree to authenticate dummy bits attached to blocks.

**Corollary 2.** *Assuming that the underlying ORAM scheme described in Section 3 has semi-honest security (i.e., replace Definition 2 in Appendix B with a semi-honest adversary $\mathcal{A}$), and that* $\mathsf{MAC}$ *is a secure message authentication scheme, then the augmented scheme with time- and location-aware message authentication codes (as described above) satisfies malicious security by Definition 2.*

*Proof.* (sketch.) The simulator is simulating interactions with a real-world adversary $\mathcal{A}$. For every block (or metadata) the semi-honest simulator intends to send to the semi-honest real-world adversary, now the simulator interacting with a malicious $\mathcal{A}$ additionally authenticates the block (or metadata) with time- and location-aware message authentication code. Whenever $\mathcal{A}$ returns a block (or metadata), the simulator verifies the message authentication code. If the verification fails, the simulator simply aborts. Effectively, any deviation of from the correct behavior by $\mathcal{A}$ can be detected except with negligible probability. Therefore, any deviation from the correct behavior translates to an aborting attack in the ideal world. □

**Cost of malicious security.** With malicious security, we need to add extra message authentication codes to data and metadata. However, it is not hard to see that while these message authentication codes increase the concrete cost, the same asymptotics are maintained.