NanoGRAM: Garbled RAM with $\widetilde{O}(\log N)$ Overhead

Andrew Park   Wei-Kai Lin   Elaine Shi[*]

Carnegie Mellon University

Abstract

We propose a new garbled RAM construction called NanoGRAM, which achieves an amortized cost of $\widetilde{O}(\lambda \cdot (W \log N + \log^3 N))$ bits per memory access, where λ is the security parameter, W is the block size, and N is the total number of blocks, and $\widetilde{O}(\cdot)$ hides $\mathsf{poly}\log\log$ factors. For sufficiently large blocks where $W = \Omega(\log^2 N)$, our scheme achieves $\widetilde{O}(\lambda \cdot W \log N)$ cost per memory access, where the dependence on N is optimal (barring $\mathsf{poly}\log\log$ factors), in terms of the evaluator's runtime. Our asymptotical performance matches even the *interactive* state-of-the-art (modulo $\mathsf{poly}\log\log$ factors), that is, running Circuit ORAM atop garbled circuit, and yet we remove the logarithmic number of interactions necessary in this baseline. Furthermore, we achieve asymptotical improvement over the recent work of Heath et al. Our scheme adopts the same assumptions as the mainstream literature on practical garbled circuits, i.e., circular correlation-robust hashes or a random oracle. We evaluate the concrete performance of NanoGRAM and compare it with a couple baselines that are asymptotically less efficient. We show that NanoGRAM starts to outperform the naïve linear-scan garbled RAM at a memory size of $N = 2^9$ and starts to outperform the recent construction of Heath et al. at $N = 2^{13}$.

Finally, as a by product, we also show the existence of a garbled RAM scheme assuming only one-way functions, with an amortized cost of $\widetilde{O}(\lambda^2 \cdot (W \log N + \log^3 N))$ per memory access. Again, the dependence on N is nearly optimal for blocks of size $W = \Omega(\log^2 N)$ bits.

1 Introduction

Garbled circuits, originally proposed by Yao [Yao86, Yao82], is a cryptographic technique for two parties to perform secure computation over their private data in two rounds. At a high level, a garbler can garble some computation expressed as a circuit as well as the inputs. An evaluator who obtains the garbled circuit and garbled inputs can securely evaluate the function over the inputs, resulting in garbled outputs that can only be decoded using the garbler's secret key. The evaluator learns nothing about the garbled inputs or outputs. Subsequently, numerous works have focused on making garbled circuits increasingly more practical [Yao86, Yao82, KS08, CKKZ12, App13, KMR14, ZRE15, RR21, HK21b, HK20, HK21a, HKO21]. In practice, however, computations are expressed in the Random Access Machine (RAM) model which is a mismatch for the circuit model. Converting RAM programs to circuits in general incur polynomial overhead in the RAM's space and time, making it prohibitive in practice especially when the computation involves big data. To avoid this expensive RAM-to-circuit conversion overhead, the elegant work of Lu and Ostrovsky [LO13] suggested a new abstraction called garbled RAM, which aims to garble a RAM program directly without converting it to a circuit. From a theoretical perspective, the goal of garbled RAM is to garble a program incurring only $\mathsf{poly}(\lambda, \log N)$ overhead where λ is the security parameter and

[*] Author ordering is randomized.

Table 1: Comparison with prior works, where S_λ denotes the circuit size of the PRF that outputs λ bits, and CCR hash is Circular Correlation-Robust hash. See Appendix B and C for details.

	Assumption	Communication (bits)	Blackbox
Lu and Ostrovsky [LO13]	Circular GC[a]	$\widetilde{O}(\lambda S_\lambda W \log^2 N)$	No
Hazay and Lilintal [HL20]	OWF	$O(\lambda S_\lambda \cdot (W \log N + \lambda \log^2 N + \log^3 N)$	No
Garg et al. [GLO15]	OWF	$\widetilde{O}\left(\lambda^2 \cdot (W \log^4 N + \log^6 N)\right)$	Yes
Heath et al. [HKO21]	CCR hashes	$O\left(\lambda \cdot (W \log^2 N + \log^4 N)\right)$	Yes
	OWF[b]	$O\left(\lambda^2 \cdot (W \log^2 N + \log^4 N)\right)$	Yes
This work	CCR hashes	$\widetilde{O}\left(\lambda \cdot (W \log N + \log^3 N)\right)$	Yes
	OWF	$\widetilde{O}\left(\lambda^2 \cdot (W \log N + \log^3 N)\right)$	Yes

a. Circularly secure garbled circuit, see [GHL+14].
b. This is not documented in their paper, but it is a standard method to tweak their scheme.

N denotes the space of the RAM. Since the original work of Lu and Ostrovsky [LO13], a line of works [GHL+14, GLO15, LO17, HKO21] have focused on improving garbled RAM constructions.

With the exception of the most recent work by Heath et al. [HKO21], prior works on garbled RAM [GHL+14, GLO15, LO17] did not care about the poly factor in the $\mathsf{poly}(\lambda, \log N)$ overhead, let alone concrete performance. Nonetheless, since garbled RAM was originally motivated by the need to speed up garbled random-access computation on big data, clearly, our dream is to make garbled RAM practical some day. The very recent work of Heath et al. [HKO21] took a pioneering step towards this dream: they constructed a garbled RAM scheme that achieves $O(\lambda \cdot (W \log^2 N + \log^4 N))$ overhead where W denotes the block size. Specifically, when the block size $W = \Omega(\log^2 N)$, their scheme achieves $O(\lambda \cdot W \cdot \log^2 N)$ overhead. Their scheme assumes the existence of a circular correlation-robust hash or a random oracle — the same assumptions as the mainstream practical garbled circuit literature, including FreeXOR [KS08, CKKZ12] and subsequent improvements [KMR14, ZRE15, RR21].

As a baseline of comparison, imagine that we actually allowed interaction. In this case, the state-of-the-art (for moderately large data) is running the Circuit ORAM algorithm [WCS15] on top of an efficient garbled circuit implementation. In this case, the overhead would be $O(\lambda \cdot (W \log N + \log^3 N))$, which is a logarithmic factor smaller than that of Heath et al. [HKO21]. In this paper, we ask the following natural question:

> *Can we have a (non-interactive) garbled RAM scheme whose asymptotical performance is competitive to the interactive state-of-the-art, that is, running Circuit ORAM on top of garbled circuits?*

Our results and contributions. We answer the above question affirmatively. Following the elegant work of Heath et al. [HKO21], we take another significant step forward towards the dream of making garbled RAM practical. Concretely, we show a new garbled RAM construction called NanoGRAM, that incurs $\widetilde{O}\big(\lambda \cdot (W \log N + \log^3 N)\big)$ overhead where $\widetilde{O}(\cdot)$ hides $\mathsf{poly} \log\log$ factors. In comparison with Heath et al. [HKO21], we save almost a logarithmic factor. Our scheme makes the same assumptions as Heath et al. [HKO21] as well as the standard literature on efficient garbled circuits [KS08, CKKZ12, KMR14, ZRE15, RR21], i.e., either assuming circular correlation-robust hashes or the random oracle model. Further, our garbled RAM construction is *blackbox* in the sense that it does not require garbling the circuit of some cryptographic primitive such as a pseudorandom function (PRF).

Theorem 1.1 (Garbled RAM from circular correlation-robust hashes)**.** *Assume circular correlation-robust hashes or the random oracle model. There is a blackbox garbled RAM scheme where each memory access incurs an amortized cost of* $\widetilde{O}\left(\lambda \cdot (W \log N + \log^3 N)\right)$ *where* λ *is the security parameter,* W *is the block size, and* N *is the total number of blocks.*

As a direct corollary, if $W = \Omega(\log^2 N)$*, then our garbled RAM scheme achieves* $\widetilde{O}(\lambda \cdot W \cdot \log N)$ *amortized cost per memory access.*

Modulo the $\mathsf{poly} \log \log$ factors, we believe that there may be some barriers for further improving our asymptotical results for blackbox garbled RAMs[1]. First, for block sizes $W = \Omega(\log^2 N)$, our scheme has *optimal* dependence on N (barring $\mathsf{poly} \log \log$ factors) due to well-known ORAM lower bounds [GO96, Gol87, LN18]. Second, for small block sizes, any further asymptotical improvement would likely imply a *statistically* secure ORAM that breaks the $O(\log^2 N)$ barrier — this is arguably the biggest open problem in the ORAM line of work, and no progress has been made for a long time[2]. Although computationally secure ORAMs [PPRY18, AKL+20] are a logarithmic factor more efficient than statistically secure ones, so far we do not know how to use computationally secure ORAM techniques in blackbox garbled RAMs, i.e., without having to garbled the PRF employed by the ORAM. Third, as mentioned, even when allowing interactions, we do not know any scheme that performs asymptotically better than the Circuit-ORAM-over-garbled-circuit baseline.

Our work also gives rise to a garbled RAM scheme from OWF but it incurs an extra λ factor in cost, as stated in the following corollary:

Corollary 1.2 (Garbled RAM from one-way functions)**.** *Assume the existence of one-way functions. There exists a garbled RAM scheme that achieves* $\widetilde{O}\left(T \cdot \lambda^2 \cdot (\log^3 N + W \log N)\right)$ *amortized cost per memory access, where* $\widetilde{O}(\cdot)$ *hides* $\mathsf{poly} \log \log \lambda$ *factors.*

In particular, for large enough blocks $W = \Omega(\log^2 N)$*, the resulting garbled RAM incurs* $\widetilde{O}\left(T \cdot \lambda^2 \cdot W \log N)\right)$ *amortized cost per memory access.*

Table 1 compares our asymptotical result with prior garbled RAM works. Besides those listed in the table, Gentry et al. [GHL+14] also propose a garbled RAM scheme from one-way function and identity-based encryption with poly-logarithmic cost. Additionally, they also propose a garbled RAM scheme from one-way function only but the asymptotical cost is N^ϵ for some constant $\epsilon \in (0, 1)$. We did not include it in the table because the result is subsumed by Garg et al. [GLO15].

Concrete performance. In Section 8, we suggest several practical optimizations to our garbled RAM scheme described in Theorem 1.1. We developed a simulator for our garbled RAM scheme with these suggested optimizations. Our simulation results show that we break even with the naïve linear scan GRAM at about $N = 2^9$ memory size, and we start to outperform the prior work EpiGRAM [HKO21] at about $N = 2^{13}$ memory size.

2 Technical Roadmap

2.1 Background

Encodings. We will use the following forms of encodings.

[1] Although using indistinguishability-based obfuscation, we can compress the size of the garbled RAM [CCC+16, CH16, CCHR16], these techniques do not save the evaluator's runtime in comparison with garbled RAM.

[2] Garbled RAM only needs an ORAM in a relaxed model where we do not charge the cost of pre-processing, but even in this relaxed model, it remains an open question how to construct a $o(\log^2 N)$ statistical ORAM.

- *Garbling.* Suppose we choose some secret key $\mathsf{sk} = \Delta = \{0,1\}^\lambda$ where λ is the security parameter. Suppose every wire, which carries one bit, is assigned a *label* (also called a *language*) $L \in \{0,1\}^\lambda$. The *garbling* of a bit $b \in \{0,1\}$ on this wire, denoted $\{\!\{b\}\!\}$, is computed as $\{\!\{b\}\!\} = \Delta \cdot b \oplus L$. This encoding approach was first proposed in the elegant Free XOR work [KS08]. For a vector of bits $x \in \{0,1\}^k$, we use $\{\!\{x\}\!\}$ to mean the garbling of each bit one by one.
- *Sharing.* For efficiency purposes, we also adopt another form of encodings called *sharings* [HKO21] that support only restricted forms of computation to be elaborated later. Given a random *label* (also called a *language*) $L \in \{0,1\}^k$, we can create a sharing $[\![x]\!]$ of a k bit string $x \in \{0,1\}^k$, that is, $[\![x]\!] = x \oplus L$.

For the time being, the reader may imagine that all encodings are in the form of garblings. We will explain how to use sharings to improve the efficiency later.

The language translation problem. In a garbled circuit scheme, every garbled gate essentially performs some garbled computation over the garbled input wires, the computation result is encoded using the language of the output wires. Since the wiring in a circuit is static, the garbler knows the mapping between each gate's output and input languages a-priori, and can prepare the garbled truth table for each gate accordingly.

As prior works observed [LO13,GHL$^+$14,GLO15,LO17,HKO21], in a garbled RAM scheme, the key challange is that of a *dynamic* language translation for a memory read or write. Take memory read for example, and henceforth, we also refer to each memory word as a *block*. Suppose that some garbled block resides at some physical location α, and is therefore garbled using a language related to the physical location α. We want to read the block back, but instead encoded using a global-time-dependent label. Only in this way, can we successfully feed this garbled block to the CPU's garbled next-instruction circuit. One can imagine that the garbler prepares a garbled next-instruction circuit for every time step t, and each such garbled circuit speaks a language dependent on the time t. The challenge is that the physical location to read in each time step t is dynamically generated, and cannot be determined statically at garbling time. This means that we need to dynamically translate location-dependent encodings to time-dependent encodings.

Switch: a minimal gadget for dynamic translation. A garbled switch, proposed in the elegant work of Heath et al. [HKO21], is a basic building block that performs dynamic translation between a parent and two children nodes. Suppose that the parent node receives some garbled data and a garbled direction bit indicating which of the two children should receive the data. The parent node now wants to re-encode the data using a language that the corresponding child recognizes, so the child can receive the data and potentially perform some garbled computation on it. The security requirement says that the evaluator cannot learn anything about the encoded data, but it is allowed to learn the direction bit. Imagine that each node keeps track of some *local time* which corresponds to the number of times the node has been invoked. When garbled data arrives at any node, the input data should be encoded using a label that depends on the node's local time. To garble such a switch, the main challenge comes from the fact that the parent and the two children have different local clocks. When the parent routes garbled data to one of the children, it must re-encode the data using a language that depends on the child's local time. Unfortunately, the garbler cannot statically predict the mapping between the parent's local time and the destination child's local time.

Informally, a garbled switch has the following abstraction:

- **Garble.** The garbler receives an array of input labels denoted $\mathbf{InL}$, and two stacks of output labels denoted $\mathbf{OutL}_0$ and $\mathbf{OutL}_1$, respectively. Specifically, $\mathbf{InL}[\tau]$ denotes the language of the τ-th invocation of the parent node, $\mathbf{OutL}_0[\tau]$ denotes the language of the τ-th invocation of the left child, and $\mathbf{OutL}_1[\tau]$ denotes the language of the τ-th invocation of the right child. The garbler then outputs some garbled circuitry GC and garbled memory Gmem to be consumed later by the evaluator.
- **Switch.** The evaluator can consume GC and Gmem to perform garbled switch operations described below. In every time step τ (of the parent), the parent receives $\{\!\{b\}\!\}$ and $\{\!\{\mathsf{data}\}\!\}$ where $b \in \{0, 1\}$ is a direction bit and data denotes the data to be routed to the b-th child. The evaluator can securely evaluate the following functionality: pop the next unconsumed label L from the b-th stack $\mathbf{OutL}_b$, re-encode data using the label L, and output the result. We allow the evaluator to learn the direction bit b, however, it should not learn anything about the garbled data data.

Heath et al. [HKO21] proposed an elegant idea that leverages two garbled stacks [ZRE15, WNL+14, HKO21] to realize a garbled switch. Specifically, the garbler initializes two garbled stacks with the encoded contents $\mathbf{OutL}_0$ and $\mathbf{OutL}_1$, respectively. Whenever a new request arrives at the parent node, the evaluator makes a real pop from the b-th stack and makes a fake pop from the $(1-b)$-th stack. The result of the real pop is an encoded label that corresponds to the current local time of the b-th child. The result of the fake pop is simply an encoding of 0. Observe that both popped values are encoded using labels dependent on the parent's local time. Similarly, the input $\{\!\{\mathsf{data}\}\!\}$ is also garbled using a label dependent on the parent's local time. This makes it possible for the garbler to prepare a garbled circuit in advance that re-encodes the input $\{\!\{\mathsf{data}\}\!\}$ using the popped label instead.

The cost of garbling such a switch is directly related to how many accesses we must provision. Suppose that each of the two children can be visited at most m times, and thus the parent can be visited at most $2m$ times. In this case, the parent's switch would need two garbled stacks each of capacity m. Using existing garbled stack techniques [ZRE15, WNL+14, HKO21], the cost is $O_\lambda(w \cdot m \log m)$ where w is the payload length (i.e., the bit width of data), and we use $O_\lambda(\cdot)$ to hide factors that depend on the security parameter λ. This directly translates to an amortized cost of $O_\lambda(w \cdot \log m)$ per switch operation. Note that later on, we will actually care about minimizing the factors that depend on λ and w; however, for ease of understanding, we ignore these factors for the time being.

Why Heath et al. [HKO21] is inefficient. At a very high level, Heath et al. [HKO21] builds upon this minimal switch gadget that is capable of dynamic translation, and eventually obtains a full garbled RAM. Their blueprint is to first use garbled switches to build an *access-revealing one-time memory*, and then upgrade the access-revealing one-time memory to a full-fledged garbled RAM through a hierarchical data structure and recursion techniques. Interestingly, their usage of the hierarchical data structure and recursion is novel and tailored specifically for garbled RAM; it makes use of the fact that the data structure performs shufflling and the garbler is aware of the data shuffling pattern ahead of time, since the garbler is choosing the random coins used in the shuffling.

There are a couple of reasons why the approach of Heath et al. [HKO21] is asymptotically and concretely non-optimal. One of the most important reasons is because their composition of garbled switches in a tree-like fashion is inefficient. To obtain an access-revealing one-time memory of size n, they need to garble a tree of switches with n leaves. The root node must provision for up to n accesses, each of the root's children must provision for $n/2$ accesses, . . ., and each leaf must provision for one access. For simplicity, assume $w \geq \log n$. The total cost to garble the tree of switches would therefore be $O_\lambda(w \cdot n \log^2 n)$; which translates to an amortized cost of $O_\lambda(w \cdot \log^2 n)$ for each single request to the one-time memory. This cost is pre-recursion. After applying the full recursion, their asymptotical cost[3] becomes $O_\lambda(W \cdot \log^2 N + \log^4 N)$.

We wish to reduce the cost by roughly a logarithmic factor, that is, we aim for $\widetilde{O}_\lambda(W \cdot \log N)$ *pre-recursion* cost per memory access where $\widetilde{O}(\cdot)$ hides $\mathsf{poly} \log\log$ factors, rather than their $O_\lambda(W \cdot \log^2 N)$ cost.

2.2 Our Approach

As mentioned, with the exception of Heath et al. [HKO21], earlier works on garbled RAM [LO13, GHL+14, GLO15, LO17] adopt a two-step compilation approach : 1) compile the RAM program to an Oblivious RAM whose memory access patterns are safe to reveal — this approach can rely on off-the-shelf Oblivious RAM algorithms [WCS15, CS17]; 2) compile an oblivious RAM to a garbled RAM (where the garbling does not shield memory accesses). Each step of the compilation incurs a separate poly-logarithmic overhead, and the two sources of overheads are multiplied. Heath et al. [HKO21] suggested a second approach where we work at a lower level of abstraction, and design customized garbled data structures and gadgets and then compose them into a Garbled RAM scheme.

First attempt. We adopt the second approach. Since a garbled RAM scheme must embed some Oblivious RAM (ORAM) scheme in it, a natural attempt is to take a state-of-the-art *statistically* secure ORAM[4] such as Circuit ORAM [WCS15, CS17], and ask how we can garble such a data structure.

We briefly describe the underlying non-recursive tree-based data structure that underlies Circuit ORAM [WCS15, CS17]. The full ORAM scheme involves creating logarithmically many such trees through a standard recursion technique [SCSL11, SSS12]. The pre-recursion ORAM tree is a binary tree with n leaf nodes, and each non-root node is a bucket of some capacity $O(1)$. The root bucket is super-logarithmic in size for storing overflowing blocks. The main *path invariant* is that every block is assigned to a random path (i.e., a path from the root to a random leaf node), and the choice of this random path is not revealed until the block is next accessed. To fetch a block, one looks up the path where the block resides through recursion, and the path can be identified by a leaf node often denoted leaf — we also call leaf the block's position identifier. Then, one looks up all buckets on the path from the root to the leaf node leaf. When a block with the requested logical address addr is encountered, the block is removed from the corresponding bucket. At this moment, the block is updated if the current operation is a write operation, and a new random path is chosen for the block. The block is then added back to the root bucket tagged with its new position identifier. After every access, we need to perform some maintenance operation that moves blocks closer to the leaf level, such that none of the buckets will overflow except with negligible probability. We may assume that the access patterns of the maintainance operations are a-priori fixed, e.g., using the reverse lexicographical order eviction idea first suggested by Gentry et al. [GGH+13].

[3] Throughout the paper, we use capitalized letters N and W to denote the number of blocks and block size of the final GRAM construction, and we use small letters n, m, and w to denote the size and payload length of building blocks. The reason for this distinction is because we need to instantiate multiple instances of these building blocks with varying parameters in the final scheme.

[4] Although *computationally* secure ORAMs can achieve asymptotically better overhead in cloud outsourcing scenarios, we currently do not know any way to use computationally secure ORAMs in *blackbox* garbled RAM schemes, without having to securely evaluate the circuits of cryptographic primitives such as pseudo-random functions.

To garble such a tree-based ORAM, a main challenge is that online phase has dynamic access patterns: every time we request a block, it goes through a random path in the ORAM tree. To solve this challenge, we can potentially rely on the garbled switch data structure. Suppose that every node in the tree has a garbled switch. When a memory access request arrives, it comes with $\{\!\!\{\mathsf{addr}, \mathsf{leaf}\}\!\!\}$ where addr is the block's logical address, and leaf is the block's position identifier; further, the request is garbled using a global-time-dependent label which also coincides with the local time of the root switch. Note that the cleartext value of leaf may be safely exposed to the evaluator. Recall that during this access, each bucket on some path will search for a block with the desired logical address addr, and if so, it returns the block's payload; else, it returns 0. We want to make sure that each bucket's fetch result is encoded using some global-time-dependent language, and the collection of all $O(\log n)$ languages are denoted $L_0, \ldots, L_{O(\log n)}$. Let $\{\!\!\{L_0, \ldots, L_{O(\log n)}\}\!\!\}$ be an encoding of these languages under some global-time-dependent label that is recognized by the root whose local clock coincides with the global clock.

Now, imagine that the root receives the information $\{\!\!\{\mathsf{addr}, \mathsf{leaf}, L_0, \ldots, L_{O(\log n)}\}\!\!\}$. It uses $b = \mathsf{leaf}[0]$ as the direction bit, and wants to route the information it has received to the b-th child. To achieve this, it must first re-encode the pair addr and leaf using a label that is dependent on the local time of the b-th child — and this can be accomplished by the garbled switch. Imagine that every node along the path does the same, and each node uses the next bit in leaf to decide its direction. In this way, each node along the path can receive a fetch instruction garbled using a language that matches its local time, and it can look in its own garbled memory whether a block exists with the desired addr. The fetch result is garbled using the corresponding garbled label which it received as part of the garbled input (i.e., $L_0, \ldots, L_{O(\log n)}$). Finally, some garbled CPU circuit can securely aggregate all $O(\log n)$ fetched results into a final result.

This naïve scheme has two sources of inefficiency. First, the root switch must provision for n accesses, each of the root's children must provision for $n/2 \pm o(n)$ accesses with high probability, and so on. Therefore, the total cost of all the switches is $O_\lambda(w \cdot \log^2 n)$ where w denotes the length of the payload being routed. The second drawback is the fact that the length of the payload w is large, since we need to route $O(\log n)$ labels each of λ bits long.

These two sources of inefficiency each incurs an extra $\log n$ factor that we want to get rid off. Below we discuss how to overcome these two sources of inefficiency. We shall begin with the second problem, which is a little easier than the first one.

2.2.1 Passing a Single Label with an XOR Trick

To overcome the second challenge, we introduce an XOR trick as depicted in Figure 1. Assume that each node in the tree has a garbled bucket henceforth denoted GBkt and a garbled switch denoted $\mathsf{GSwitch}$. A garbled bucket GBkt supports a Read operation: when given a logical address $\{\!\!\{\mathsf{addr}\}\!\!\}$ garbled under an local-time-dependent input label, it will output the corresponding block's contents $\{\!\!\{\mathsf{val}\}\!\!\}$ if the block is found, or output $\{\!\!\{0\}\!\!\}$ if not found. Further, the result is garbled using a local-time-dependent output label. Suppose that we want the final memory fetch result to be encoded under some global-time-dependent label K. Henceforth assume that the root is at level 0 of the tree, and let $\ell_{\max} = O(\log n)$ be the leaf level. As we traverse the path, each non-leaf bucket along the way encodes its result using labels $L_0, L_1, \ldots, L_{\ell_{\max}-1}$, respectively (we abuse notations where $L_0, L_1, \ldots$ are now *local-time-dependent*). Our idea is to pass an encoding of the label $L_{\ell_{\max}} = K \oplus L_0 \oplus \ldots \oplus L_{\ell_{\max}-1}$ to the leaf node, such that the leaf bucket will encode its fetch result using the label $L_{\ell_{\max}}$. This way, all the labels would XOR to K. This means that when we XOR the garbling of all $\ell_{\max}+1$ fetched results, we obtain a garbling of the fetched result encoded under the label K. To achieve this, we can have each node in a non-leaf level ℓ pass an encoding

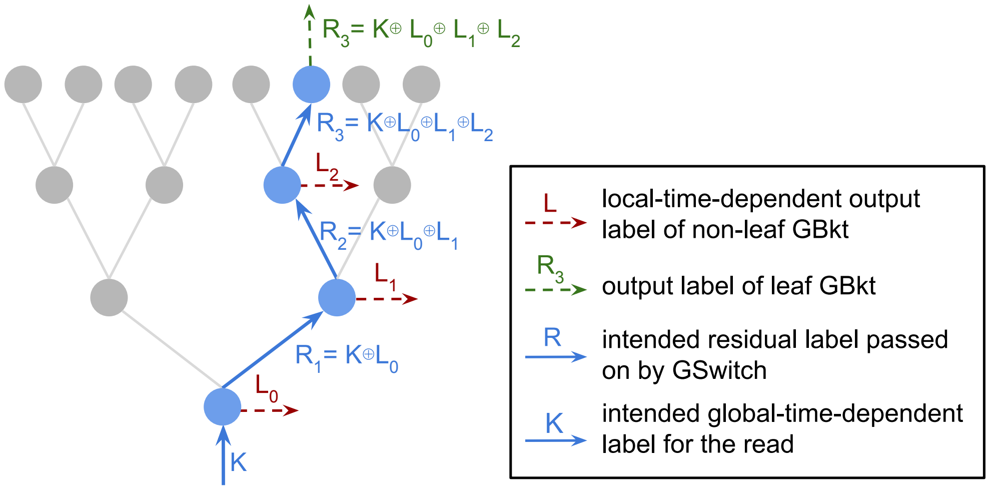

R3= K⊕ L0⊕ L1⊕ L2
R3= K⊕L0⊕L1⊕L2
L2
R2= K⊕L0⊕L1
L1
R1= K⊕L0
L0
K
L
local-time-dependent output label of non-leaf GBkt
R3
output label of leaf GBkt
R
intended residual label passed on by GSwitch
K
intended global-time-dependent label for the read

Figure 1: XOR trick.

of the residual label $R_\ell = K \oplus L_0 \oplus \ldots \oplus L_{\ell-1}$ to its child, encoded using a language dependent on the child's local time. The XOR trick saves us one logarithmic factor in cost.

2.2.2 Splitting Switches into Poly-logarithmically Sized Ones

To overcome the first challenge, our idea is to avoid using big switches that must be provisioned with a large number of accesses. Instead, we want to break up the big switches into poly-logarithmically sized ones. To achieve this, we observe that we can leverage ideas from the Bucket ORAM algorithm [FNR+15].

Background on Bucket ORAM. At a very high level, Bucket ORAM is a tree-based ORAM but with a hierarchical-style rebuild algorithm.

Let T be the maximum runtime of the RAM program, and let N be its space. In the Bucket ORAM tree, each bucket has size $2\mathsf{B} = O(\log(\frac{T\cdot N}{\delta}))$ where δ is the statistical failure probability. Like in any tree-based ORAM scheme [SCSL11], a bucket can store either *filler* blocks denoted $\perp$ or *real* blocks of the format $(\mathsf{addr}, \mathsf{leaf}, \mathsf{data})$ where addr is the block's logical address, leaf denotes its position identifier, and data denotes its payload. The read phase of the algorithm is also like any tree-based ORAM [SCSL11, SvDS+13, WCS15, CS17]. To read a block, we first recursively look up its position identifier denoted leaf, we then look up the path from the root leading to leaf for the block requested. The block is removed from the corresponding bucket if found. Besides the tree data structure, there is also a small stash that can store up to B blocks. Any memory request must also search in the stash for the desired block. Moreover, after a block is fetched, it will be added to the stash (possibly with an updated payload string). For the time being, one can imagine that each bucket itself as well as the stash implement small ORAMs [DMN11, CNS18, CSLN21] such that they can look up a block in $\mathsf{poly}\log\log(\frac{T\cdot N}{\delta})$ time.

Interestingly, the maintainance phase of Bucket ORAM actually resembles a hierarchical ORAM [GO96, Gol87]. Suppose that n and B are powers of 2. Let root be at level 0, and let $\ell_{\max} = \log_2 \frac{n}{\mathsf{B}}$. Each level i is rebuilt every $2^i \cdot \mathsf{B}$ steps. In particular, at the end of some time step t, if $t+1$ is a multiple of n, we need to rebuild levels $0, \ldots, \ell_{\max}$ into level $\ell_{\max}$ and empty all remaining levels. Else, if we can express $t+1$ as $j \cdot 2^\ell$ for some odd integer j, then we need to rebuild levels $0, \ldots, \ell-1$ into level ℓ, emptying the levels $0, \ldots, \ell-1$ in the process. Further, the rebuild process must respect

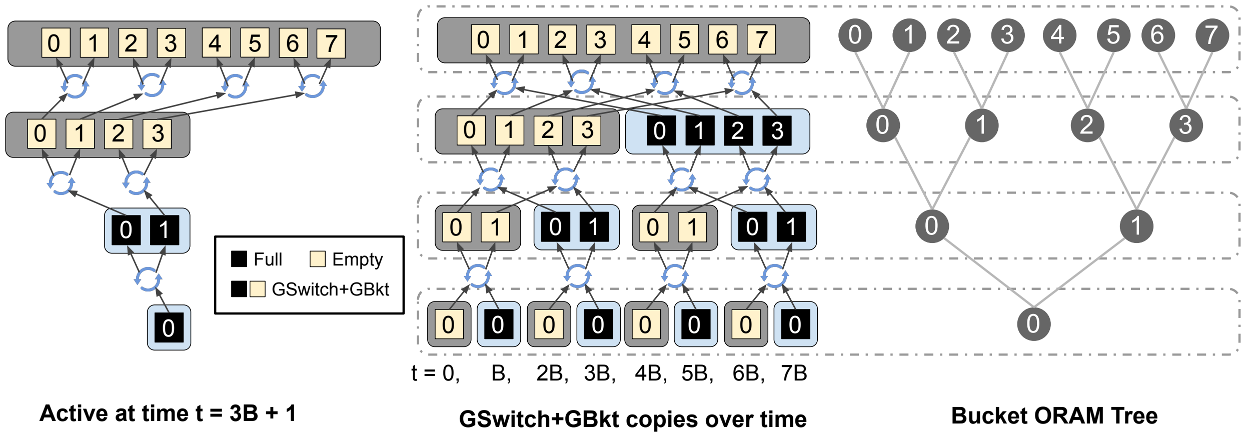

0 1 2 3 4 5 6 7
0 1 2 3
0 1
0
Full
Empty
GSwitch+GBkt
Active at time t = 3B + 1
t = 0, B, 2B, 3B, 4B, 5B, 6B, 7B
GSwitch+GBkt copies over time
Bucket ORAM Tree

Figure 2: Each node at level ℓ has $T/(\mathsf{B} \cdot 2^{\ell})$ copies of $\mathsf{GSwitch} + \mathsf{GBkt}$, and at time t, the $\lfloor t/(\mathsf{B} \cdot 2^{\ell}) \rfloor$-th copy is active. The numbers show which copies of garbled circuitry correspond to which tree node in the same level.

the position identifier each block has chosen. The Bucket ORAM work [FNR+15] shows how to accomplish this rebuild using a circuit whose size is *linear* in total number of elements involved in the rebuilding. For the purpose of this work, the details of the rebuild algorithm is not too important. Therefore, we give a brief description below and refer the reader to the Bucket ORAM work [FNR+15] for details. At a high level, the Bucket ORAM work suggested that this rebuild can be accomplished through a sequence of MergeSplit operations. In each MergeSplit operation, we take a pair of buckets as inputs and and output a pair of buckets. Each real block in the input buckets will go into one of the output buckets, and the choice depends on the corresponding bit in their leaf label. The MergeSplit operation essentially relies on sorting of objects with 1-bit keys, i.e., compaction [AKL+20]. Indeed, if we use a linear-sized compaction circuit to realize each MergeSplit, the total cost of the rebuild would be linear. For our paper, it does not matter to our final asymptotics even if we used bitonic sort to implement the MergeSplit, since this part of the overhead will not be the dominating factor.

Splitting switches into poly-logarithmically sized ones. As shown in Figure 2, each node at level ℓ in the tree has $T/(2^{\ell} \cdot \mathsf{B})$ instances of GBkt and GSwitch. The instances are indexed from $0, 1, \ldots, T/(2^{\ell} \cdot \mathsf{B}) - 1$. During time step $t \in [0 : T)$, the garbled instances indexed $\lfloor t/(2^{\ell} \cdot \mathsf{B}) \rfloor$ are active. Whenever a level is rebuilt, the existing GBkt and GSwitch instances corresponding to all tree nodes in this level finalize, and new instances are initialized.

Due to the rebuild schedule of Bucket ORAM, we know in advance for each instance at some parent node, which instances of its children it must communicate with. In other words, the communication graph between the instances are statically determined.

There are, however, some subtle challenges we need to resolve for this idea to work. Observe that half the switches finalize together with their children — this case is a little easier to handle since the new instances that take over can start fresh. For the other half, when they finalize, their children do not finalize at the same time. However, their children's local clocks have already advanced to some dynamic value which cannot be predicted in advance. In this case, we need to implement an explicit *hand-over* operation such that the new switches can inherit the necessary states from the switches whose jobs they are taking over. To achieve this we need the help of garbled data structures supporting dynamic finalization which we explain below.

2.2.3 Garbled Data Structures with Dynamic Finalization

We adopt a modular framework to present our scheme which makes it easier to verify its correctness and security. A new abstraction we propose is a garbled data structure with a dynamic finalization — we believe that our definitions may be of independent interest in future works on garbled data structures and algorithms.

Consider some data structure that supports some function calls $\mathsf{Func}_1, \ldots, \mathsf{Func}_c$. Additionally, there is a special function called $\mathsf{Finalize}$ which is called at the end of its life cycle to output some final garbled state — for example, the final garbled state can be an encoding of all unvisited blocks stored in the data structure. We assume that except for the $\mathsf{Finalize}$ function, the call schedule for all other functions are fixed a-priori. The $\mathsf{Finalize}$ function, however, may be called at any time t^* within some a-priori known time bound $t_{\max}$. No matter in which local time step t^* the function $\mathsf{Finalize}$ is invoked, the finalized states it outputs must be garbled under some fixed label (that does not depend on t^*). To enforce that the evaluator calls $\mathsf{Finalize}$ at the right time, the $\mathsf{Finalize}$ call has to take in a garbled signal $\{\!\{1\}\!\}$ that explicitly authorizes the call. More specifically, a garbled data structure supporting dynamic finalization has the following abstraction:

- **Garbler**. The garbler takes in some initial memory array $\mathbf{DB}$, input and output labels denoted $\mathbf{InL}$ and $\mathbf{OutL}$, and outputs the garbled circuit GC and initial garbled memory Gmem. Specifically, $\mathbf{InL}$ and $\mathbf{OutL}$ provide the following labels:

$$\begin{aligned}\mathbf{InL} &:= (I_0, \ldots, I_{t_{\max}-1}, C_0, \ldots, C_{t_{\max}-1}, C_{t_{\max}}) \\ \mathbf{OutL} &:= (O_0, \ldots, O_{t_{\max}-1}, F)\end{aligned}$$

  where for $\tau \in [0 : t_{\max})$, I_τ and O_τ denote the time-dependent labels used to encode the input and the output of the τ-th (non-$\mathsf{Finalize}$) operation, respectively; for $t^* \in [0 : t_{\max}]$, C_{t^*} is the label used to encode the finalization signal should $\mathsf{Finalize}$ be invoked at time step t^*; and F denotes the label used to encode the final state st output by $\mathsf{Finalize}$.

- **Evaluator.**

  1. In each local time step $\tau \in [0 : t_{\max})$, the evaluator can call garbled operations $\{\!\{\mathsf{outp}\}\!\} \leftarrow \mathsf{Func}_{i_\tau}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{inp}\}\!\})$ where the call schedule $i_\tau \in [c]$ is fixed a-priori. The inputs and outputs must be garbled under labels dependent on the local time. The operations may cause updates to the internal garbled memory.
  2. At some dynamic point of time $t^* \in [0 : t_{\max}]$, the evaluator may call $\widetilde{\mathsf{st}} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem}, \widetilde{1})$: The evaluator must input a garbled finalization signal $\widetilde{1}$ (which is garbled under a t^*-dependent label). Intuitively, this signal forces the evaluator to evaluate $\mathsf{Finalize}$ in the intended time step t^* and not any other time step. The $\mathsf{Finalize}$ algorithm outputs a garbled final state denoted $\widetilde{\mathsf{st}}$, which is garbled under the fixed label F which is *independent* of t^*.

Garbled data structures with dynamic finalization are used in multiple places in our construction. For example,

- Each GBkt instance is visited a dynamic number of times before finalization, and when finalized, it must output the remaining unvisited elements encoded under some fixed label. The results will then be passed to the garbled rebuilder algorithm.
- Each $\mathsf{GSwitch}$ instance is also visited a dynamic number of times just like GBkt. As mentioned earlier, for half of the switches, when they finalize, they must pass some internal state to the next switch that takes over, such that the next switch knows the local clocks of the children.

- Finally, some of the building blocks (e.g., garbled stack, access-revealing one-time memory) we use to construct our GBkt and $\mathsf{GSwitch}$ are also garbled data structures with dynamic finalization.

We formally define the security for such garbled data structures with dynamic finalization in Section 3.3, and we give efficient instantiations partly relying on a building block called an expiring vault (see Section 5.1).

The need to support dynamic finalization complicates our construction. In several cases, we cannot use existing building blocks for this reason and have to construct our own variants. For example, in our construction, each GBkt itself is a small garbled dictionary capable of translating a memory fetch result from using a location-based label to using a local-time-dependent label. Since we need a dynamic finalization capability from the GBkt, we cannot directly use prior work such as Heath et al. [HKO21]. Similarly, for other seemingly standard building blocks such as garbled stack, we also have to construct our own variants and prove them secure.

2.2.4 Additional Optimizations

So far, we have explained our ideas assuming that all wires are encoded using garbling. To save a factor of λ, we adopt several ideas suggested by Heath et al. [HKO21]. In particular, we will encode some wires using sharings rather than garblings. Unlike garblings, sharings are space-preserving since the sharing of some string has the same length as the original string. However, sharings can only be involved in restricted computations.

1. a shared bit can be XORed with another shared bit or a constant value known at garbling time that is hard-wired in the garbled circuitry or garbled memory, and the outcome of such an operation is a sharing too, i.e., $(\llbracket x \rrbracket, \llbracket y \rrbracket) \rightarrow \llbracket x \oplus y \rrbracket$;
2. a shared string may be multipled with a garbled bit whose cleartext value is known by the evaluator, and the result of the operation is a sharing, i.e., $\left(\{\!\{b^E\}\!\}, \llbracket y \rrbracket\right) \rightarrow \llbracket b \cdot y \rrbracket$ where $y \in \{0,1\}^k$. Throughout the paper, if the evaluator is allowed to know the cleartext of some garbled value $\{\!\{\mathsf{val}\}\!\}$, we often write $\{\!\{\mathsf{val}^E\}\!\}$ to make this explicit.

The elegant work by Heath et al. [HKO21] described techniques to efficiently implement the above operations involving shared bits, assuming the existence of a random oracle. Specifically, the first type of operations require only 1 bit per XOR gate, the second type of operations require only $O(k + \lambda)$ bits to garble a gate that multiply $\{\!\{b^E\}\!\}$ with $\llbracket y \rrbracket$ where k is the bit-width of y.

Later on in our constructions, the data stored in garbled stacks which are part of the garbled switches will be in the form of sharings; furthermore, the labels passed long the tree paths will also be in the form of sharings. These optimizations save us a λ factor in the final costs.

3 Preliminaries and Definitions

3.1 Labels, Encoding and Decoding

Let $\mathsf{sk} := \Delta \in \{0,1\}^\lambda$ be some global secret that is randomly chosen upfront by calling some key generation algorithm $\mathsf{Gen}(1^\lambda)$. Δ is also the secret key of the garbler. We will often use the following types of encodings of bits or bitstrings:

- **Garbling** $\{\!\{b\}\!\}$: for some bit $b \in \{0,1\}$, let $\{\!\{b\}\!\} := \Delta \cdot b \oplus L$ where $L \in \{0,1\}^\lambda$ denotes a (pseudo-)random string. We often call L the *label*[5] used to garble $\{\!\{b\}\!\}$, or write $L = \mathsf{Lbl}(\{\!\{b\}\!\})$.

[5] Prior works on garbled circuits often refer to the encodings for 0 and 1 (i.e., L and $\Delta \oplus L$, resp.) as the two labels for a particular wire (i.e., bit). In our paper, we refer to L as the *label*, and we refer to L and $\Delta \oplus L$ as *encodings* of

Given a string $x \in \{0,1\}^k$ of k bits, we often use the shorthand notation $\{\!\{x\}\!\}$ to denote the bit-by-bit garbling of x.

- **Sharing** $[\![x]\!]$: for a bit-string $x \in \{0,1\}^k$, let $[\![x]\!] := x \oplus L$ where $L \in \{0,1\}^k$ is a (pseudo-)random string. We often call L the *label* used to create the sharing $[\![x]\!]$, or write $L = \mathsf{Lbl}([\![x]\!])$.

Notice that unlike a garbling, a sharing does not blow up the length of the string that is shared. The introduction of sharing can improve the efficiency of garbled circuits as shown in prior work [HKO21]. As mentioned later, unlike garblings, sharings can only be involved in restricted types of computations.

Decoding. Knowing the secret key $\mathsf{sk} := \Delta$ and the label $L \in \{0,1\}^\lambda$ used to encode some garbling $\{\!\{b\}\!\}$, it is easy to decrypt the bit $b \in \{0,1\}$ that is encoded. Similarly, knowing the label $L \in \{0,1\}^k$ used to encode some sharing $[\![x]\!]$, it is easy to decrypt the bit $x \in \{0,1\}^k$ that is encoded.

3.2 Garbled Circuits

Garbled circuits was first proposed in the original work of Yao [Yao86, Yao82]. Since then, a line of works have improved its practical efficiency [KS08, CKKZ12, App13, KMR14, ZRE15, RR21, HK21b, HK20, HK21a, HKO21]. In our paper, since we allow two types of encodings, garbling and sharing. Garbled bits can be paired up with each other and be involved either AND or XOR gates. However, we assume that there are some restrictions on the type of computations that shared bits can be involved in:

1. a shared bit can be XORed with another shared bit or a constant value known at garbling time that is hard-wired in the garbled circuitry or garbled memory, and the outcome of such an operation is a sharing too, i.e., $([\![x]\!], [\![y]\!]) \rightarrow [\![x \oplus y]\!]$;
2. a shared string may be multipled with a garbled bit whose cleartext value is known by the evaluator, and the result of the operation is a sharing, i.e., $\left(\{\!\{b^E\}\!\}, [\![y]\!]\right) \rightarrow [\![b \cdot y]\!]$ where $y \in \{0,1\}^k$. Throughout the paper, if the evaluator is allowed to know the cleartext of some garbled value $\{\!\{\mathsf{val}\}\!\}$, we often write $\{\!\{\mathsf{val}^E\}\!\}$ to make this explicit.

One can mechanically verify that all the circuits we need to garble in our constructions respect the above assumptions regarding the use of shared bits. The prior work by Heath [HKO21] described techniques to efficiently implement the above operations involving shared bits, assuming the existence of a random oracle. Specifically, the first type of operations is free, and the second type of operation requires only $O(k + \lambda)$ bits to garble a gate that multiply $\{\!\{b^E\}\!\}$ with $[\![y]\!]$ where k is the bit-width of y.

For deriving our asymptotical bounds, it is sufficient to assume that any AND or XOR gate that involves only garbled bits take $O(\lambda)$ bits to encode. In a practical implementation, however, one should use the more recent techniques such as Free-XOR [KS08, CKKZ12], FleXOR [KMR14], half-gates [ZRE15], and subsequent improvements [RR21, HK21b, HK20, HK21a, HKO21] to improve the concrete performance. Note that we adopt the encoding scheme originally proposed in the Free-XOR work [KS08, CKKZ12].

0 and 1, respectively. Note that once we fix the $\mathsf{sk} = \Delta$ and the label L for some wire, both encodings for 0 and 1 are determined.

3.3 Garbled Data Structure

Our building blocks involve several garbled data structures. An evaluator can invoke multiple garbled operations of the data structure during its life cycle. Every garbled data structure has a *local time* denoted $\tau \in [0 : t_{\max}]$ where $t_{\max}$ is the maximum number of operations supported. When the τ-th operation is called, we say that the garbled data structure is in local time τ. Unless otherwise stated, our garbled data structures will have the following interface where we use $\widetilde{x}$ to denote an encoding of x which is either a garbling or sharing of x:

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{DB}, \mathbf{InL}, \mathbf{OutL})$: the algorithm takes in the security parameter, some secret key $\mathsf{sk} \in \{0,1\}^\lambda$, parameters params (explained shortly), the initial memory array $\mathbf{DB}$, input and output labels denoted $\mathbf{InL}$ and $\mathbf{OutL}$ used to encode the garbled inputs and outputs respectively. It outputs the garbled memory Gmem and some garbled circuits denoted GC. Here, the parameters params typically contains the word size often denoted w, the length (often denoted m) of the initial memory array $\mathbf{DB}$, and the maximum number of operations denoted $t_{\max}$.
- $\mathsf{Gmem}', \widetilde{\mathsf{outp}} \leftarrow \mathsf{Func}_1^{\mathsf{GC}}(\mathsf{Gmem}, \widetilde{\mathsf{inp}})$,
  $\ldots$,
  $\mathsf{Gmem}', \widetilde{\mathsf{outp}} \leftarrow \mathsf{Func}_c^{\mathsf{GC}}(\mathsf{Gmem}, \widetilde{\mathsf{inp}})$: some functions to be called by the evaluator. We assume that the *call schedule for the functions* $\mathsf{Func}_1, \ldots, \mathsf{Func}_c$ *is known a-priori*, where the call schedule specifies exactly which of these functions will be invoked in each time step $\tau \in [0 : t_{\max})$. For the evaluator to evalute these functions in a garbled manner, it needs to consume the garbled circuitry GC which we write in the superscript of the procedure. Calling these garbled operations not only outputs some encoded answer $\widetilde{\mathsf{outp}}$, but also may result in updates to the internal encoded memory denoted Gmem'. The inputs $\widetilde{\mathsf{inp}}$ and outputs $\widetilde{\mathsf{outp}}$ are garbled using labels dependent on the data structure's local time.
- $\widetilde{\mathsf{st}} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem}, \widetilde{1})$: the $\mathsf{Finalize}$ function can be invoked in *any* time step $t^* \in [0 : t_{\max}]$, where t^* also denotes the number of operations invoked prior to calling $\mathsf{Finalize}$. Unless otherwise noted, *exactly when* $\mathsf{Finalize}$ *will be invoked is unknown at the time of garbling.* To successfully invoke $\mathsf{Finalize}$, the evaluator must input a garbled finalization signal $\widetilde{1}$ (which is garbled under a t^*-dependent label). Intuitively, this signal forces the evaluator to evaluate $\mathsf{Finalize}$ in the intended time step t^* and not any other time step. The $\mathsf{Finalize}$ algorithm outputs a garbled final state denoted $\widetilde{\mathsf{st}}$, which is garbled under a fixed label which is *independent* of t^*.

The input/output labels $\mathbf{InL}$ and $\mathbf{OutL}$ fed into the Garble algorithm should contain the following:

$$\begin{aligned} \mathbf{InL} &:= (I_0, \ldots, I_{t_{\max}-1}, C_0, \ldots, C_{t_{\max}-1}, C_{t_{\max}}) \\ \mathbf{OutL} &:= (O_0, \ldots, O_{t_{\max}-1}, F) \end{aligned}$$

where for $\tau \in [0 : t_{\max})$, I_τ and O_τ denote the time-dependent labels used to encode the input inp and the output outp in the τ-th time step, respectively; for $t^* \in [0 : t_{\max}]$, C_{t^*} is the label used to encode the finalization signal should $\mathsf{Finalize}$ be invoked at time step t^*; and F denotes the label used to encode the final state st output by $\mathsf{Finalize}$.

Relationship with garbled circuits. Garbled circuits can be viewed as a special case of our garbled data structure formulation. Specifically, a garbled circuit can be viewed as a garbled data structure that supports only one operation Func after garbling. For this reason, we do not give a separate definition for garbled circuits. Later on, we will rely on garbled circuits as a building block to construct garbled data structures.

Correctness. Suppose that there is some (insecure) data structure $\mathcal{DS}$ supporting the operations $f_1, \ldots, f_c$ and fin. We say that a garbled data structure scheme correctly implements $\mathcal{DS}$ iff for any $\lambda \in \mathbb{N}$, any $\mathsf{sk} \in \{0,1\}^\lambda$, any $\mathsf{params} = (m, w, t_{\max})$, any $\mathbf{DB}$, any $\mathbf{InL}$ and $\mathbf{OutL}$, any $1 \leq t^* \leq t_{\max}$, any sequence of function calls $i_0, \ldots, i_{t^*-1} \in [c]$, any input sequence $\mathsf{inp}_0, \ldots, \mathsf{inp}_{t^*-1}$: let $\mathsf{outp}_0, \ldots, \mathsf{outp}_{t^*-1}, \mathsf{st}$ be the correct outcomes when we initialize $\mathcal{DS}$ with $\mathbf{DB}$ and then make the calls $\{f_{i_\tau}(\mathsf{inp}_\tau)\}_{\tau \in [0:t^*)}$, and fin in sequence, then, the following must be true with probability 1:

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{DB}, \mathbf{InL}, \mathbf{OutL})$;
- for $\tau \in [0 : t^*)$: let $\widetilde{\mathsf{inp}_\tau}$ be a correct encoding of inp_τ using label I_τ, let $\mathsf{Gmem}, \widetilde{\mathsf{outp}_\tau} \leftarrow \mathsf{Func}_{i_\tau}^{\mathsf{GC}}(\mathsf{Gmem}, \widetilde{\mathsf{inp}_\tau})$;
- let $\widetilde{1}$ be a correct encoding of the finalization signal 1 under label C_{t^*}, let $\widetilde{\mathsf{st}} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem}, \widetilde{1})$;
- then, it must be that $\{\widetilde{\mathsf{outp}_\tau}\}_{\tau \in [0:t^*)}$ and $\widetilde{\mathsf{st}}$ are valid encodings of the correct outputs $\{\mathsf{outp}_\tau\}_{\tau \in [0:t^*)}$ and st, under the labels $\{O_\tau\}_{\tau \in [0:t^*)}$ and F, respectively.

Security. We define the security of garbled data structures beolow.

Definition 1 (Security of garbled data structures (and garbled circuits))**.** We say that a garbled data structure scheme is secure w.r.t. some leakage function $\mathsf{Leak}(\cdot)$, iff there exists probabilistic polynomial-time (p.p.t.) simulators Sim, such that for for any $\lambda \in \mathbb{N}$, any $\mathsf{params} = (m, w, t_{\max})$, any $\mathbf{DB}$, any $1 \leq t^* \leq t_{\max}$, any sequence of function calls $i_0, \ldots, i_{t^*-1} \in [c]$, for any input sequence $\mathsf{inp}_0, \ldots, \mathsf{inp}_{t^*-1}$, any output labels $\mathbf{OutL}$ of appropriate length, for any subset of inputs $S \subseteq \{\mathsf{inp}_\tau, 1_\tau\}_{\tau \in [0:t^*)}$ whose encodings are to be simulated, for any choice of $\mathbf{InL}[\neg S]$ where $\mathbf{InL}[\neg S]$ denotes the part of $\mathbf{InL}$ used to encode the set $\neg S$, the outputs of the real and ideal experiments below are computationally indistinguishable:

Real experiment. Input: λ, params, $\mathbf{DB}$, t^*, function calls $i_0, \ldots, i_{t^*-1} \in [c]$, input sequence $\mathsf{inp}_0, \ldots, \mathsf{inp}_{t^*-1}$, subset of inputs S, subset of input labels $\mathbf{InL}[\neg S]$, output labels $\mathbf{OutL}$.

1. Sample $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)$, and sample the remaining unspecified input labels $\mathbf{InL}[S]$ at random;
2. Let $\{\widetilde{S}, \widetilde{\neg S}\} = \{\widetilde{\mathsf{inp}_\tau}, \widetilde{1_\tau}\}_{\tau \in [0:t^*)}$ be correctly encoded inputs and finalization signals using sk and labels $\mathbf{InL}$;
3. Let Gmem, $\mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda$, sk, params, $\mathbf{DB}$, $\mathbf{InL}$, $\mathbf{OutL})$;
4. Output $\mathsf{Gmem}, \mathsf{GC}, \widetilde{S}$.

Ideal experiment. Input: λ, params, $\mathbf{DB}$, t^*, function calls $i_0, \ldots, i_{t^*-1} \in [c]$, input sequence $\mathsf{inp}_0, \ldots, \mathsf{inp}_{t^*-1}$, subset of inputs S, subset of input labels $\mathbf{InL}[\neg S]$, output labels $\mathbf{OutL}$.

1. Sample $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)$;
2. Let $\widetilde{\neg S}$ be correctly encoded inputs in subset $\neg S$ using sk and labels $\mathbf{InL}[\neg S]$;
3. Run the ideal functionality using the given $\mathbf{DB}$, function calls $f_{i_0}, \ldots, f_{i_{t^*-1}}$, and input sequence $\mathsf{inp}_0, \ldots, \mathsf{inp}_{t^*-1}$, and finally, run fin. Let $\mathsf{outp}_0, \ldots, \mathsf{outp}_{t^*-1}$ and st be the results correspondingly;

4. Let $\{\widetilde{\mathsf{outp}_\tau}\}_{\tau\in[0:t^*)}$ and $\widetilde{\mathsf{st}}$ be correctly encoded outputs and finalized state using sk and labels $\mathbf{OutL}$;
5. Run the simulator

$$\mathsf{Sim}\left(1^\lambda, \mathsf{params}, t^*, \{i_\tau, \widetilde{\mathsf{outp}}_\tau\}_{\tau\in[0:t^*)}, \widetilde{\mathsf{st}}, \widetilde{\neg S}, \mathsf{Leak}(\{i_\tau, \mathsf{inp}_\tau\}_{\tau\in[0:t^*)})\right)$$

and output the result.

Note that when $\neg S = \emptyset$, then the above notion is a direct adaptation of the standard security definition for garbled circuits to garbled data structures. Therefore, the above definition can be viewed as a generalization of standard garbled circuit security. In particular, this generalization allows us to fix the encodings of a subset of the inputs denoted $\neg S$, feed these encodings $\widetilde{\neg S}$ to the simulator, and have the simulator simulate the the rest of the garbled inputs $\widetilde{S}$, along with the garbled circuitry GC and garbled memory Gmem. We sometimes refer to the set of inputs $\neg S$ whose input labels have been fixed as the *fixed set*, and the set of inputs S whose input labels are not fixed as the *free set*. We make this generalization for convenience later. Jumping ahead, when we write our garbled algorithms, we often allow *garbled input sharing*, that is, the same garbled input wire is fed into two or more garbled components. In this case, we will need to use the generalized security definition in our proofs.

As mentioned earlier, we have two forms of encodings, garblings and sharings. Later in our constructions, in fact only *garbled* wires (as opposed to shared wires) can be input to multiple garbled components. Therefore, we additionally impose the following constraints to Definition 1:

- The fixed set $\neg S$ must contain only *garbled* inputs variables;
- Any *shared* input must be in the free set S.

Existing constructions of garbled circuits [Yao86,Yao82,KS08,CKKZ12,App13,KMR14,ZRE15, RR21], including the techniques needed from Heath et al. [HKO21] naturally satisfy the above generalized notion too.

Encoding cleartext outputs. Later in our construction, sometimes we also have a garbled circuit or garbled data structure output cleartext rather than encoded outputs. Our above formulation actually also captures cleartext outputs if we use the encoding scheme described in Section 3.1, and thus we can adopt this formulation without loss of generality. In particular, a cleartext output bit can be expressed as either a sharing whose label is 0, or a garbling whose label ends with a 0 bit. In particular, we will follow the approach mentioned in prior work [ZRE15,HKO21], where we choose Δ at random subject to the last bit being 1. In this way, as long as the label of a garbling ends with a 0 bit, the last bit of the encoding will be the cleartext value of the bit.

Performance metric. We will use the size of the garbled program, measured in terms of the number of bits, as our performance metric.

Remark 1. Unless otherwise noted, we assume the above syntax and conventions for any garbled data structure we define. There is one slight exception, which is the data structure $\mathsf{GSwitch}$ defined in Section 5.3 — in fact, this is the critical data structure for handling the non-determinism of memory accesses. Jumping ahead a little, $\mathsf{GSwitch}$ is initialized with two stacks of output labels denoted $\mathbf{OutL}_0$ and $\mathbf{OutL}_1$, and every operation, one label is popped from a stack of choice, and this popped label will be used as the output label.

Remark 2. Known garbled circuits constructions [Yao86, Yao82, KS08, CKKZ12, App13, KMR14, ZRE15, RR21, HK21b, HK20, HK21a, HKO21] also satisfy the following notion of simulation — our proofs also make use of this simulation notion. There exists a simulator Sim', such that for any output labels **OutL**,

$$\mathsf{GC} \overset{c}{\equiv} \mathsf{Sim}'(1^\lambda, C)$$

where GC is the honest garbling of the circuit C using randomly generated input labels as well as **OutL**, and $\overset{c}{\equiv}$ means computational indistinguishability. This notion says we can simulate the garbled circuitry without knowing the (encoded) outputs, if we do not have to also simulate the active encoded inputs.

3.4 Notational Conventions

Omitting Gmem and GC without risking ambiguity. In the above, we use Gmem to denote a garbled data structure's internal encoded memory. Since the external caller of the data structure need not worry about Gmem, when we write our algorithms, we often omit writing the Gmem term explicitly. Moreover, we also omit writing the GC in the superscript of the garbled function calls without risking ambiguity. For example, suppose we use $\mathsf{GDataStruct}$ to denote some instance of a garbled data structure, we often write $\widetilde{\mathsf{outp}} \leftarrow \mathsf{GDataStruct.Func}_i(\widetilde{\mathsf{inp}})$, omitting the Gmem as well as the GC-superscript. This means that this function call is consuming the Gmem and GC of the $\mathsf{GDataStruct}$ instance.

Implicit label matching convention. We often rely on an implicit label matching convention to describe our garbled data structures. For example, if we write the following statements as part of the evaluator's algorithm:

```
$\mathsf{GDataStruct}_0.\mathsf{Func}(\{\!\{x\}\!\})$ :
    $\{\!\{y\}\!\} \leftarrow \mathsf{GDataStruct}_1.\mathsf{Foo}(\{\!\{x\}\!\})$;
    $\{\!\{z\}\!\} \leftarrow \mathsf{GDataStruct}_2.\mathsf{Bar}(\{\!\{y\}\!\})$;
    output $\{\!\{z\}\!\}$;
```

Assuming that $\mathsf{GDataStruct}_1$ and $\mathsf{GDataStruct}_2$ are not called anywhere else, then the above implies that

- the input label of the τ-th call to $\mathsf{GDataStruct}_0.\mathsf{Func}$ should match the the input label of the τ-th call to $\mathsf{GDataStruct}_1.\mathsf{Foo}$;
- the output label of the τ-th call to $\mathsf{GDataStruct}_1.\mathsf{Foo}$ should match the the input label of the τ-th call to $\mathsf{GDataStruct}_2.\mathsf{Bar}$;
- the output label of the τ-th call to $\mathsf{GDataStruct}_0.\mathsf{Func}$ should match the the output label of the τ-th call to $\mathsf{GDataStruct}_2.\mathsf{Bar}$;

Unless otherwise noted, the labels for all variables are randomly selected subject to such implicit matching constraints (which can always be unambiguously implied by our algorithm description).

4 Building Blocks

4.1 Garbled Stash (GStash)

Let $\mathsf{params} = (m, w)$, where m is the number of times the $\mathsf{GStash.Read}$ and $\mathsf{GStash.Add}$ functions will be invoked before the data structure is finalized, and w is the bit width of each block's payload

val. The call schedule is a-priori fixed: the data structure will receive Read and Add calls in an interleaved fashion, and there will be a total of m calls to each function. For GStash, we assume that its $\mathsf{Finalize}$ function will always be invoked at an *a-priori fixed* time, that is, at time $t = m$, where t is the number of Add calls that have completed so far (not including the current call if we are now inside an Add call). This explains why the $\mathsf{Finalize}$ function *need not take any garbled signal as input.* A GStash data structure has the following interface:

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{InL}, \mathbf{OutL})$: takes in the parameters params and input/output labels $\mathbf{InL}$ and $\mathbf{OutL}$, and outputs the garbled circuitry GC and initial garbled memory Gmem. We often write $\mathbf{OutL} := (\mathbf{RdL}, \mathbf{FinL})$. where $\mathbf{RdL}$ denotes the labels used by the output of Read and $\mathbf{FinL}$ denotes the labels used by the output of $\mathsf{Finalize}$.
- $\mathsf{Gmem}', \{\!\{\mathsf{val}\}\!\} \leftarrow \mathsf{Read}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{addr}\}\!\})$: takes in a requested garbled address $\{\!\{\mathsf{addr}\}\!\}$, and if a block with the requested addr is found, val should be the correct value of the block; else $\mathsf{val} = 0$. Further, $\mathsf{Lbl}(\mathsf{val}) = \mathbf{RdL}[t]$ where t denotes the current time step.
- $\mathsf{Gmem}' \leftarrow \mathsf{Add}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{addr}, \mathsf{val}\}\!\})$: receives a garbled block which contains its own logical address $\{\!\{\mathsf{addr}\}\!\}$ and a payload string $\{\!\{\mathsf{val}\}\!\}$, and stores the block in the data structure;
- $\{\!\{\mathsf{stash}\}\!\} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem})$: this function is always invoked at time m. It outputs $\{\!\{\mathsf{stash}\}\!\}$ which contains all the garbled blocks that have been added but have not been read afterwards; further, $\mathsf{Lbl}(\mathsf{stash})$ should match $\mathbf{FinL}$.

Construction. We will adopt a naïve construction for GStash. We shall create a length-m garbled array which is initialized with all $\{\!\{0\}\!\}$s. Every time step, to read an element we simply use a garbled linear scan circuit which scans through the entire array to look for the requested element. To add an element, we simply write it into position t of the garbled array. Upon finalization, the final garbled array is output. This naïve scheme costs $O(\lambda w m)$ bits.

4.2 Slow BlackBox Garbled Dictionary (GDict)

A garbled dictionary implements a standard static dictionary interface. First, the data structure is initialized with an initial array $\mathbf{DB}$. Each entry of $\mathbf{DB}$ is either a *filler* denoted $\perp$, or a *real* element of the form $(\mathsf{key}, \mathsf{val})$. Next, one can make multiple Lookup requests where each request asks for some key. If an entry with key is found in $\mathbf{DB}$, then return the corresponding val; else, return $\perp$.

Let $\mathsf{params} = (m, \mathbf{w}, t_{\max})$ where m denotes the number of entries in the initial array $\mathbf{DB}$, $\mathbf{w}$ contains the bit-widths of key and val, respectively, and $t_{\max}$ is the maximum number of Lookup requests. A GDict has the following interface.

- $\mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{InL}, \mathbf{OutL})$: receives the input and output labels $\mathbf{InL}$ and $\mathbf{OutL}$, and outputs GC and Gmem;
- $\mathsf{Gmem} \leftarrow \mathsf{Init}^{\mathsf{GC}}(\{\!\{\mathbf{DB}\}\!\})$: receives an initial array $\{\!\{\mathbf{DB}\}\!\}$ and outputs the initial garbled memory Gmem;
- $\mathsf{Gmem}', \{\!\{\mathsf{val}\}\!\} \leftarrow \mathsf{Lookup}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{key}\}\!\})$: receives a request $\{\!\{\mathsf{key}\}\!\}$, if key exists in $\mathbf{DB}$, then output a garbling of the corresponding value $\{\!\{\mathsf{val}\}\!\}$; else, output $\{\!\{\perp\}\!\}$. Regardless of which case, the garbling should use $\mathbf{OutL}[\tau]$ where τ is the local time, i.e., the total number of Lookup requests so far (not counting the current one).

Construction. We can adopt a binary search tree and garble it using the blackbox garbled RAM scheme proposed by Heath et al. [HKO21]. Recall that the garbled RAM scheme by Heath et al. [HKO21] costs $O(\lambda(w \log^2 m + \log^4 m))$ bits per memory access to implement a garbled memory of size m and block size w. The cost of GDict is $O(t_{\max} \cdot \lambda \cdot (w_1 \log^3 m + w_2 \log^2 m + \log^5 m))$ where $w_1 = |\mathsf{key}|$ and $w_2 = |\mathsf{val}|$.

If we use the standard oblivious data structure technique [WNL$^+$14] and piggyback the binary search tree's index structure on top of the recursion data structure of Heath et al. [HKO21], the cost of the GDict can be further reduced to $O(t_{\max} \cdot \lambda \cdot (w_2 \cdot \log^2 m + (w_1 + \log m) \cdot \log^3 m))$.

4.3 Garbled Random Permutation (GPerm)

A garbled random permutation circuit implements the following functionality: takes an input array and outputs a random permutation of it. In the garbled version, not only must the inputs and outputs be hidden, but the permutation itself also must be hidden. Although permutation is efficiently realizable by a circuit, GPerm implements a *randomized* functionality. Therefore, we explicitly define the abstraction below. Let $\mathsf{params} = (m, w)$ where m denotes the number of entries of $\mathbf{DB}$, and w denotes the bit-width of each entry. A garbled random permutation denoted GPerm has the following syntax.

- $\mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{InL}, \mathbf{OutL})$: receives the intended input and output labels and outputs a garbled circuit GC;
- $\{\!\{\mathbf{DB'}\}\!\} \leftarrow \mathsf{Perm}^{\mathsf{GC}}(\{\!\{\mathbf{DB}\}\!\})$: takes a garbled input array $\{\!\{\mathbf{DB}\}\!\}$ and outputs a garbled permutation of the input denoted $\{\!\{\mathbf{DB'}\}\!\}$.

We require the following security and correctness definition (combined in a single definition). We say that GPerm implements a garbled random permutation algorithm, iff there exists a p.p.t. simulator Sim, such that for any $\lambda \in \mathbb{N}$, any $\mathsf{params} = (m, w)$, any $\mathbf{DB}$, any output labels $\mathbf{OutL}$ of appropriate length,

$$\left(\{\!\{\mathbf{DB'}\}\!\},\ (\mathsf{GC}, \{\!\{\mathbf{DB}\}\!\})\right) \overset{c}{\equiv} \left(\{\!\{\mathcal{F}^{\mathrm{perm}}(\mathbf{DB})\}\!\}^{\langle \mathbf{OutL} \rangle},\ \mathsf{Sim}\left(1^\lambda, \mathsf{params}, \{\!\{\mathcal{F}^{\mathrm{perm}}(\mathbf{DB})\}\!\}^{\langle \mathbf{OutL} \rangle}\right)\right)$$

where $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)$, $\mathbf{InL} \overset{\$}{\leftarrow} \{0,1\}^{|\mathbf{InL}|}$, let $\{\!\{\mathbf{DB}\}\!\}$ be an honest garbling of $\mathbf{DB}$ using sk and labels $\mathbf{InL}$, let $\mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{DB}, \mathbf{InL}, \mathbf{OutL})$, and let $\{\!\{\mathbf{DB'}\}\!\} \leftarrow \mathsf{Perm}^{\mathsf{GC}}(\{\!\{\mathbf{DB}\}\!\})$. In the above $\mathcal{F}^{\mathrm{perm}}(\cdot)$ is an ideal functionality that outputs a random permutation of the input array it receives, and the notation $\{\!\{x\}\!\}^{\langle \mathbf{OutL} \rangle}$ means a garbling of each bit of x using the secret key sk and labels $\mathbf{OutL}$.

Construction. We can use Waksman's permutation network [Wak68] which is a network of $m \log m$ swap gates. The garbler picks a random permutation at garbling time, and this decides the swap-or-not decision (henceforth called control bit) for every gate. The garbler can garble the permutation network, and create garbled states for the control bits of all gates. It is not hard to see that this construction satisfies the above security definition. The cost of GPerm is $O(\lambda w m \log m)$.

5 Building Blocks for Garbled Memory

5.1 Expiring Vault (GVault)

An expiring vault is a simple garbled data structures used to store up to $t_{\max}$ secrets, henceforth denoted $S_0, \ldots, S_{t_{\max}-1}$. In every time step, either an empty procedure denoted Skip is called, or the $\mathsf{Finalize}$ procedure is invoked with some authenticated finalization signal. If $\mathsf{Finalize}$ is invoked at time τ with an authenticated finalization signal garbled under the current time, then it outputs a subset of the secrets $S_\tau, \ldots, S_{t_{\max}-1}$.

Let $\mathsf{params} = (t_{\max}, w)$ where $t_{\max}$ denotes the maximum number of time steps, and w denotes the width of the secrets. An expiring vault GVault has the following syntax:

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, (S_0, \ldots, S_{t_{\max}-1}), \mathbf{InL})$: takes in the security parameter 1^λ, the secret key sk, the parameters $\mathsf{params} = (t_{\max}, w)$ which contains the maximum time $t_{\max}$ and the length of each secret w, a list of secret values $(S_0, \ldots, S_{t_{\max}-1})$, the input labels $\mathbf{InL} = (C_0, \ldots, C_{t_{\max}})$, and outputs Gmem and GC;
- $\mathsf{Gmem}', \perp \leftarrow \mathsf{Skip}^{\mathsf{GC}}(\mathsf{Gmem}, \perp)$: an empty function that merely increments the local time;
- $\mathsf{res} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{1\}\!\})$: if the input $\{\!\{1\}\!\}$ is a correct garbling of 1 under C_τ where τ is the current time, then the output res should be $S_\tau, \ldots, S_{t_{\max}-1}$.

Construction. The garbler creates $\{\{\!\{1\}\!\}_\tau\}_{\tau \in [0:t_{\max})}$ garblings of the bit 1 under the $t_{\max}$ input labels of $\mathsf{Finalize}$, respectively. Henceforth[6], let $k_\tau := \{\!\{1\}\!\}_\tau$. For each $\tau \in [0 : t_{\max} - 2]$, the garbler uses k_τ to encrypt S_τ and $k_{\tau+1}$, and let c_τ, c'_τ be the resulting ciphertexts. Finally, the garbler uses $k_{t_{\max}-1}$ to encrypt $S_{t_{\max}-1}$, and let $c_{t_{\max}-1}$ be the resulting ciphertext. The resulting $\mathsf{GC} := (\{c_\tau, c'_\tau\}_{\tau \in [0:t_{\max}-2]}, c_{t_{\max}-1})$, and $\mathsf{Gmem} := \perp$. When the evaluator obtains $k_\tau = \{\!\{1\}\!\}_\tau$, it is easy to see that it can decrypt all of $S_\tau, \ldots, S_{t_{\max}-1}$. The cost of the above GVault construction is $O(t_{\max} \cdot (\lambda + w))$.

The security of this construction is easy to see. We can construct a simulator in a straightforward manner. Suppose the $\mathsf{Finalize}$ function is called at time t^*. The simulator is given the inputs $S_{t^*}, \ldots, S_{t_{\max}-1}$; it may or may not be given the garbled input $\{\!\{1\}\!\}_{t^*}$ (see the generalized notion of simulation in Definition 1). If the simulator is not given the input $\{\!\{1\}\!\}_{t^*}$, it simply samples $\{\!\{1\}\!\}_{t^*}$ at random. Now, for any $\tau \geq t^*$, it computes c_τ and c'_τ honestly. For any $\tau < t^*$, it outputs simply outputs encryptions of 0 under random keys. The simulator outputs all these encryptions, and if it is not given $\{\!\{1\}\!\}_{t^*}$ as input, it also outputs the $\{\!\{1\}\!\}_{t^*}$ it has chosen. It is easy to see that the simulated view is computationally indistinguishable from the real-world view.

5.2 Stack (GStack)

5.2.1 Definition

A garbled stack GStack is initialized with some initial memory array denoted $\mathbf{DB}$, and it supports Pop operations controlled by a flag denoted $b \in \{0, 1\}$. If $b = 0$, nothing will be popped, and if $b = 1$ an element will be popped from the stack. In our application later, it is actually safe to reveal the control flag b. For GStack, we let $\mathsf{params} = (m, w, t_{\max})$, where m is the number of entries in the initial $\mathbf{DB}$, w is the bit-width of each entry, and $t_{\max}$ is the maximum number of Pop operations. It is promised that at most m number of Pop calls will have the flag b set to 1, i.e., the stack will never deplete. We shall assume that m is a power of 2, and moreover, $m \geq 16$.

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{DB}, \mathbf{InL}, \mathbf{OutL})$: takes in the security parameter λ, the parameters params, the initial stack elements $\mathbf{DB}$ containing m elements each of size w, and the

[6] Throughout the paper, we often use the notations $\{\!\{\mathsf{var}\}\!\}_\tau$ or $[\![\mathsf{var}]\!]_\tau$ to explicitly denote the fact that var pertains to the τ-th operation (e.g., it is either an input, output, or internal variable used in the τ-th operation); moreover, the variable is either garbled or shared under a τ-dependent label. When the context is clear, we may also omit the subscript τ.

Evaluator	**Garbler**
• $\mathsf{Pop}(\{\!\{b^E\}\!\})$: 1. $[\![\mathsf{res}]\!] \leftarrow \mathsf{GSlowQueue}_0.\mathsf{Pop}(\{\!\{b^E\}\!\})$;	for $\ell \in [0 : \ell_{\max}]$, call $\mathsf{GSlowQueue}_\ell.\mathsf{Garble}$ (see Equation (2) and narrative)
2. $\{\!\{\mathsf{cnt}\}\!\} \leftarrow \mathsf{GAdd}_\tau(\{\!\{\mathsf{cnt}\}\!\}, \{\!\{b^E\}\!\})$; // $\mathsf{cnt} \leftarrow \mathsf{cnt} + b$ $\{\!\{\mathsf{lcnt}_0\}\!\} \leftarrow \mathsf{GAdd}_\tau(\{\!\{\mathsf{lcnt}_0\}\!\}, \{\!\{b^E\}\!\})$; 3. let 2^ℓ be the maximum power of 2 that divides $\tau + 1$ where τ is the local time;	create the initial garbled state $\{\!\{\mathsf{cnt} = 0\}\!\}$, for $i \in [0, \ell_{\max} - 1]$, initialize $\{\!\{\mathsf{lcnt}_i = 0\}\!\}$, and for $\tau \in [0 : t_{\max})$, create the garbled circuit GAdd_τ;
4. for i from 0 to $\max(\ell_{\max} - 1, \ell)$, – $\{\!\{\beta_i\}\!\}, \{\!\{\mathsf{lcnt}_i\}\!\} \leftarrow \mathsf{GNeedUpd}_{\tau,i}(\{\!\{\mathsf{lcnt}_i\}\!\})$; /* *if* $\mathsf{lcnt}_i \geq 2$, *then* $\beta_i \leftarrow 1$ *and* $\mathsf{lcnt}_i \leftarrow \mathsf{lcnt}_i - 2$; *otherwise* $\beta_i \leftarrow 0$ */ – $[\![x_i, y_i]\!] \leftarrow \mathsf{GSlowQueue}_{i+1}.\mathsf{Pop}(\{\!\{\beta_i\}\!\})$; $\{\!\{\mathsf{lcnt}_{i+1}\}\!\} \leftarrow \mathsf{GAdd}_\tau(\{\!\{\mathsf{lcnt}_{i+1}\}\!\}, \{\!\{\beta_i\}\!\})$; – $\mathsf{GSlowQueue}_i.\mathsf{Push}(\{\!\{\beta_i\}\!\}, [\![x_i]\!], [\![y_i]\!])$; 5. call $\mathsf{GVault.Skip}()$ and return $[\![\mathsf{res}]\!]$.	for $\tau \in [0 : t_{\max})$, for each 2^i that divides $\tau + 1$, create the garbled circuit $\mathsf{GNeedUpd}_{\tau,i}$;
• $\mathsf{Finalize}(\{\!\{1\}\!\})$: 1. $\{\!\{0\}\!\}_\tau, \ldots, \{\!\{0\}\!\}_{t_{\max}-1} \leftarrow \mathsf{GVault.Finalize}(\{\!\{1\}\!\})$; 2. for $j \in [\tau : t_{\max})$, call $\mathsf{Pop}(\{\!\{0\}\!\}_j)$;	call $\mathsf{GVault.Garble}$ with $\{\{\!\{0\}\!\}_\tau\}_{\tau \in [0:t_{\max})}$ where $\{\!\{0\}\!\}_\tau$ is a garbling of 0 using the input label of the τ-th Pop call;
3. $\{\!\{\mathsf{ucnt}\}\!\} \leftarrow \mathsf{GUnary}(\{\!\{\mathsf{cnt}\}\!\}_{t_{\max}-1})$; 4. return $\{\!\{\mathsf{ucnt}\}\!\}$;	create a single garbled circuit GUnary which converts binary representation to unary representation;

Figure 3: GStack algorithm.

input/output garbling labels denoted **InL** and **OutL** respectively, and outputs some internal garbled memory Gmem and garbled circuitry GC.

- $\mathsf{Gmem}', [\![\mathsf{res}]\!] \leftarrow \mathsf{Pop}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{b^E\}\!\})$: depending on the flag b, either pop an element from the stack or do nothing. Correctness requires that 1) if $b = 1$, then the result $\mathsf{res} = \mathbf{DB}[\mathsf{cnt}_\tau]$ where τ is the current time step, and cnt_τ denotes the total number of Pop operations so far (not counting the current one) where the flag $b = 1$; and 2) if $b = 0$, then the result $\mathsf{res} = 0$. Moreover, it must be that $\mathsf{Lbl}([\![\mathsf{res}]\!])$ is the τ-th output label contained in **OutL**.

- $\{\!\{\mathsf{ucnt}\}\!\} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{1\}\!\})$: upon receiving a garbled signal $\{\!\{1\}\!\}$ indicating that the data structure should be finalized in this time step, output a garbling of ucnt, the total number of elements popped expressed in a unary format and prepended with 0s to a length of m. Correctness also requires that $\mathsf{Lbl}(\{\!\{\mathsf{ucnt}\}\!\})$ is the finalization label contained in **OutL**.

5.2.2 Construction

Although efficient garbled stacks have been proposed in earlier works [ZE13, ZRE15, WNL+14, HKO21], we need a variant that supports dynamic finalization. Therefore, we extend prior schemes with this additional feature. Our GStack algorithm employs logarithmically many garbled linear-scan queues $\mathsf{GSlowQueue}$ described in Appendix A.1, and an expiring vault GVault described in Section 5.1. Write $m = 2^\alpha$, and let $\ell_{\max} = \alpha - 3$. When the garbler receives the inputs $\mathbf{DB}, \mathbf{InL}, \mathbf{OutL}$, it parses the initial $\mathbf{DB} := (D_0, D_1, \ldots, D_{\ell_{\max}})$ as $\ell_{\max} + 1$ levels where level 0 contains 8 elements, and for $\ell \in [1 : \ell_{\max}]$, level ℓ contains $4 \cdot 2^\ell$ elements. Henceforth, for each $\ell \in [1 : \ell_{\max}]$, we may view level ℓ has having 4 wide elements of width $2^\ell \cdot w$ instead (i.e., we group 2^ℓ elements into one wide element). It will then create logarithmically many garbled slow queues:

$$\text{call } \mathsf{GSlowQueue}_0.\mathsf{Garble}(1^\lambda, \mathsf{sk}, (8, w, t_0 = t_{\max} + t_{\max}/2), D_\ell, \mathbf{IL}_\ell, \mathbf{OL}_\ell); \tag{1}$$

$$\text{for } \ell \in [1 : \ell_{\max}], \text{ call } \mathsf{GSlowQueue}_\ell.\mathsf{Garble}(1^\lambda, \mathsf{sk}, (4, 2^\ell \cdot w, t_0/2^\ell), D_\ell, \mathbf{IL}_\ell, \mathbf{OL}_\ell) \tag{2}$$

The input labels $\mathbf{IL}_\ell$ and output labels $\mathbf{OL}_\ell$ are otherwise chosen at random, with the constraints that the output labels of $\mathsf{GSlowQueue}_0.\mathsf{Pop}$ should match the output labels of $\mathsf{GStack.Pop}$, and for each $i \in [1 : \ell_{\max}]$, the output label of the τ-th call to $\mathsf{GSlowQueue}_i.\mathsf{Pop}$ should match the input labels of the τ-th call to $\mathsf{GSlowQueue}_{i-1}.\mathsf{Push}$. The remainder of the GStack algorithm is described in Figure 3. Note also that the input labels to $\mathsf{GVault.Finalize}$ should match the input labels to $\mathsf{GStack.Finalize}$. Other implicit label alignment can directly be inferred from Figure 3 (see Section 3.4 for an explanation of our implicit label matching convention).

It is not hard to see that our GStack construction costs $O(t_{\max} \cdot (w + \lambda) \cdot \log m)$ bits, if we use the efficient garbling for multiplying a garbled bit known to the evaluator with a shared value (see Section 3.2).

5.2.3 Security Proof

Theorem 5.1. *Assume that the underlying* $\mathsf{GSlowQueue}$ *and* GVault *constructions are secure by Definition 1. Then, the* GStack *construction is secure by Definition 1.*

Proof. The proof is rather mechnical. Suppose the $\mathsf{Finalize}$ function is called at time t^*. In reverse topological order of the dependency graph, we can replace the real-world garbled components one by one with simulated ones. When multiple garbled components share garbled inputs, we will need to use our generalized notion of simulation (see Definition 1), i.e., a subset of the garbled inputs may be fixed and fed into the simulator Sim which simulates the garbled circuitry/memory, as well as the remaining garbled or shared inputs.

Consider the following sequence of hybrid experiments.

- Hyb_0: the real-world view. More specifically, in the real world, we first fix the initial stack $\mathbf{DB}$ and the inputs $\{b_\tau\}_{\tau \in [0:t^*)}$. We fix all the output labels $\mathbf{OutL}$. It is possible that some of the input labels for the control bits $\{b_\tau\}_{\tau \in [0:t^*)}$ or for the finalization signals in all time steps are fixed. For the part of $\mathbf{InL}$ that are not fixed (i.e., the free set), we sample those labels at random. We then run the honest garbling algorithms and output the resulting GC, Gmem, as well as the garblings of the free set of inputs.

  In the real world, we can use the resulting GC and Gmem to perform evaluation on the garblings of the chosen $\{b_\tau\}_{\tau \in [0:t^*)}$ as well as $\{\!\{1\}\!\}_{t^*}$. The encoding of any intermediate or final variable derived from this honest evaluation is henceforth said to be an *active* encoding. By correctness of the algorithm, the active encoding of every variable must agree with the result of applying the encoding algorithm to the cleartext value of this variable, using the variable's chosen labels.

- Hyb_1: We first run Hyb_0, which generates all the active encodings for all intermediate and final variables. We then run the simulator $\mathsf{GUnary.Sim}$ which takes in the active garbled output $\{\!\{\mathsf{ucnt}\}\!\}$, and outputs $\mathsf{GUnary.GC}$ and the active encoding $\{\!\{\mathsf{cnt}\}\!\}_{t_{\max}-1}$. Given the active encoding

$\{\!\{\mathsf{cnt}\}\!\}_{t_{\max}-1}$ and the cleartext value of this variable, we can uniquely determine the label used to encode this variable (which is needed by the honest garbler $\mathsf{GAdd.Garble}$). At this moment, we can call the honest garbler to garble the remaining garbled components, and output the result.

In the remaining hybrid steps, we shall replace the garbled components one by one just like this — without risking ambiguity, henceforth we simply write the order in which we do these swaps. The details can be expanded just like the above.

- Hyb_2: otherwise identical as Hyb_1 except that 1) we simulate the terms $\llbracket\mathsf{res}\rrbracket_{t^*}, \ldots, \llbracket\mathsf{res}\rrbracket_{t_{\max}}$ at random; and 2) we replace $\mathsf{GSlowQueue}_0.\mathsf{GC}$, $\mathsf{GSlowQueue}_0.\mathsf{Gmem}$, the free set of variables among $\{\{\!\{b\}\!\}_\tau\}_{\tau\in[0:t_{\max}]}$, $\{\{\!\{\beta_0\}\!\}\}_{\tau\in[0:t_{\max}]}$, and $\{\llbracket x_0, y_0\rrbracket_\tau\}_{\tau\in[0:t_{\max}]}$ with the output of $\mathsf{GSlowQueue}_0.\mathsf{Sim}$.
- $\mathsf{Hyb}_3 = \mathsf{Hyb}_4^0$: otherwise identical as Hyb_2 except that we now replace $\mathsf{GVault.GC}$, $\mathsf{GVault}_0.\mathsf{Gmem}$ and $\{\!\{1\}\!\}_{t^*}$ (if it is a free input) with the output of $\mathsf{GVault.Sim}$.
- Hyb_4^j for $j = 1$ to $\ell_{\max}-1$: otherwise identical as Hyb_4^{j-1} except that we now replace $\mathsf{GSlowQueue}_j.\mathsf{GC}$, $\mathsf{GSlowQueue}_j.\mathsf{Gmem}$, and $\{\{\!\{\beta_j\}\!\}, \llbracket x_j, y_j\rrbracket_\tau\}_{\tau\in[0:t_{\max}]}$ with the outcome of $\mathsf{GSlowQueue}_j.\mathsf{Sim}$. Note that we need the generalized notion of simulation here because the $\{\!\{\beta_{j-1}\}\!\}_\tau$ part of the input to $\mathsf{GSlowQueue}_j.\mathsf{Pop}$ call is already fixed.
- $\mathsf{Hyb}_4^{\ell_{\max}}$: otherwise identical as $\mathsf{Hyb}_4^{\ell_{\max}-1}$ except that we now replace the terms $\mathsf{GSlowQueue}_{\ell_{\max}}.\mathsf{GC}$, $\mathsf{GSlowQueue}_{\ell_{\max}}.\mathsf{Gmem}$, and $\{\{\!\{\beta_{\ell_{\max}}\}\!\}\}_{\tau\in[0:t_{\max}]}$ with the outcome of $\mathsf{GSlowQueue}_{\ell_{\max}}.\mathsf{Sim}$.
- Hyb_5: otherwise identical as $\mathsf{Hyb}_4^{\ell_{\max}}$ except that we now replace $\{\mathsf{GNeedUpd}_{\tau,i}.\mathsf{GC}\}_{\tau\in[0:t_{\max}]}$ and $\{\{\!\{\mathsf{lcnt}_i\}\!\}_\tau\}_{\tau\in[0:t_{\max}]}$ with the output of $\{\mathsf{GNeedUpd}_{\tau,i}.\mathsf{Sim}\}_{\tau\in[0:t_{\max}]}$.
- Hyb_6: otherwise identical as Hyb_5 except that we replace $\{\mathsf{GAdd}_\tau.\mathsf{GC}\}_{\tau\in[0:t_{\max}]}$ with the output of $\{\mathsf{GAdd}_\tau.\mathsf{Sim}\}_{\tau\in[0:t_{\max}]}$. We need the generalized notion of simulation here because the garbled input $\{\!\{\mathsf{cnt}_\tau\}\!\}$, $\{\!\{b\}\!\}_\tau$, $\{\!\{\mathsf{lcnt}_i\}\!\}_\tau$ to GAdd_τ are already fixed.

□

Hyb_1 is computationally indistinguishable from Hyb_0 due to the security of garbled circuits. Hyb_2 is computationally indistinguishable from Hyb_1 due to the security of $\mathsf{GSlowQueue}_0$. Hyb_3 is computationally indistinguishable from Hyb_2 due to the security of GVault. For $j \in [1 : \ell_{\max}]$ Hyb_4^j is computationally indistinguishable from Hyb_4^{j-1} due to the security of $\mathsf{GSlowQueue}_j$. Hyb_5 is computationally indistinguishable from $\mathsf{Hyb}_4^{\ell_{\max}}$ due to the security of garbled circuits. Hyb_6 is computationally indistinguishable from Hyb_5 due to the security of garbled circuits.

Finally, observe that the experiment Hyb_6 no longer calls any real-world garbler; it only needs to call the simulators of the various garbled building blocks. Therefore, the experiment Hyb_6 actually does not need to make use of the cleartext input values any more, all it needs is the encodings of the active outputs, as well as the encodings of the fixed set of inputs. Therefore, Hyb_6 directly gives our simulator construction and the simulated view.

5.2.4 A Slightly Different Variant of GStack

Later on in our GSwitch construction, we also need a simpler variant of GStack that does not need a Finalize function — this is the variant that was constructed and proven in prior work [ZE13,HKO21]. This variant can be constructed in a similar manner as Figure 3, except that we no longer need the GVault and the Finalize function. The security proof can be done in a similar manner as the proof in Section 5.2.3.

5.3 Switch ($\mathsf{GSwitch}$)

A switch is a two-way router. Imagine that the switch receives some message $\mathsf{msg} := (\mathsf{leaf}, \mathsf{addr}, L)$. The first bit of leaf, that is, $\mathsf{leaf}[0]$, is used to determine whether the message is supposed to be forwarded to its left child or right child. The switch has a hard-wired array denoted $\mathbf{RdL}$ of length $t_{\max}$, where $t_{\max}$ is the maximum number of times that the switch can be invoked. The switch wants to route the transformed message $(\mathsf{leaf}[1:], \mathsf{addr}, L \oplus \mathbf{RdL}[\tau])$ to the child selected by $\mathsf{leaf}[0]$, where τ is the switch's local time step, i.e., how many times it has been invoked before (not including the current invocation). Later on, every node in the ORAM tree will employ such a switch to pass on information to one of its two children during an ORAM fetch operation. Altogether, this allows us to read and remove a block along a path from the root to some leaf node. In particular, each node consumes the next bit in the leaf identifier to determine the routing direction. The term $\mathbf{RdL}[\tau]$ is the local-time-dependent output label used by the garbled bucket paired with the switch, and we want to xor the incoming label L with $\mathbf{RdL}[\tau]$ before passing it on — see Section 2.2.1 for a more detailed explanation.

When we want to garble a switch, the main challenge is that of *label translation*: the input $\widetilde{\mathsf{msg}} = (\{\!\{\mathsf{leaf}, \mathsf{addr}\}\!\}_\tau, [\![L]\!]_\tau)$ is encoded using a local-time-dependent label where τ denotes the local time of the switch. The switch needs to re-encode the transformed message $(\mathsf{leaf}[1:], \mathsf{addr}, L \oplus \mathbf{RdL}[\tau])$ under a label that is dependent on the child's local time. However, the child's local time cannot be predicted at the garble time, since it depends on the actual inputs leaf which are chosen dynamically online. We adapt an elegant idea proposed by Heath et al. [HKO21] to solve this problem. Suppose that we are promised that each child will be invoked at most $t'_{\max}$ number of times. We will create two garbled stacks each containing $t'_{\max}$ labels (denoted $\mathbf{OutL}_0$ and $\mathbf{OutL}_1$, respectively), corresponding to the languages of the left and right children each time they are invoked. Given the direction bit $b := \mathsf{leaf}[0]$, we securely pop the next label from b-th stack, and we use this popped label to re-encode the output message to be routed to the corresponding child. Later in our application, we are actually allowed to leak the leaf part of the input which is related to the memory access patterns. More specifically, leaf actually corresponds to a path in the Bucket ORAM tree [FNR+15], and since its choice is random, it is safe to reveal leaf.

5.3.1 Definition

For $\mathsf{GSwitch}$, we define $\mathsf{params} = (\mathsf{B}, \mathbf{w})$ where $2\mathsf{B}$ is the maximum number of times Switch can be invoked, and $\mathbf{w}$ records the lengths of of the inputs to Switch, including the lengths of leaf, addr, and L. The lengths of all other variables will be determined by λ, $\mathbf{w}$, and B. Specifically, $\mathbf{RdL}$ contains $2\mathsf{B}$ entries each of $|L|$ bits long; $\mathbf{InL}$ contains $2\mathsf{B}$ entries each of $\lambda(|\mathsf{leaf}| + |\mathsf{addr}|) + |L|$ bits long; for $b \in \{0, 1\}$, $\mathbf{OutL}_b$ contains $2\mathsf{B}$ entries each of $|\mathbf{InL}|$ bits long; and FinL contains $2\mathsf{B}$ entries each of $2\mathsf{B}\lambda$ bits long if $\mathbf{InitL} = \emptyset$, else it contains $2\mathsf{B}$ entries each of 2λ bits long.

For each $b \in \{0, 1\}$, it is promised that Switch will only be invoked at most B times with the direction bit $\mathsf{leaf}[0] = b$. Later in our application, in fact, we guarantee that in expectation, Switch is invoked only B times, and the probability that there will be $2\mathsf{B}$ or more invocations is negligibly small. A garbled switch $\mathsf{GSwitch}$ consists of the following possibly randomized algorithms:

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{RdL}, \mathbf{InL}, \mathbf{OutL}_0, \mathbf{OutL}_1, \mathbf{FinL})$: the Garble algorithm takes in the security parameter 1^λ, the secret key sk, parameters params a list of labels $\mathbf{RdL}$ to be consumed in each time step (by the associated garbled bucket), the input labels $\mathbf{InL}$, two lists of output labels $\mathbf{OutL}_0$ and $\mathbf{OutL}_1$, as well as labels denoted FinL used to encode the output of $\mathsf{Finalize}$. It outputs the garbled circuits GC and the initial garbled memory Gmem.

We often write $\mathbf{InL} := (\mathbf{InitL}, \mathbf{ReqL}, \mathbf{CtrlL})$ where the part $\mathbf{InitL}$ is consumed by Init, the part $\mathbf{ReqL}$ is consumed by Switch, and the part $\mathbf{CtrlL}$ is used to garble the finalization signals for all time steps.

- $\mathsf{Gmem}' \leftarrow \mathsf{Init}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\!\{\mathsf{st}^E\}\!\!\})$: this function may be called at most once upfront before any invocation of Switch. Specifically, if we parse $\mathbf{InL} := (\mathbf{InitL}, \_, \_)$ where $\mathbf{InL}$ was passed to Garble, Init should be invoked if $\mathbf{InitL} \neq \emptyset$; else it will not be invoked.
- $\mathsf{Gmem}', \{\!\!\{\mathsf{leaf}'\}\!\!\}, \{\!\!\{\mathsf{addr}'\}\!\!\}, [\![L']\!] \leftarrow \mathsf{Switch}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\!\{\mathsf{leaf}^E\}\!\!\}, \{\!\!\{\mathsf{addr}\}\!\!\}, [\![L]\!])$: for correctness, the outputs must satisfy: $\mathsf{leaf}' = \mathsf{leaf}[1:]$, $\mathsf{addr}' = \mathsf{addr}$, $L' = L \oplus \mathbf{RdL}[\tau]$ where τ is the current local time step. Moreover, let $b = \mathsf{leaf}[0]$, and let cnt_b be the number of times Switch has been invoked with direction bit $\mathsf{leaf}[0] = b$ (not counting the current one); then, it must be that $\mathsf{Lbl}(\{\!\!\{\mathsf{leaf}'\}\!\!\}, \{\!\!\{\mathsf{addr}'\}\!\!\}, [\![L']\!])$ is the first $|\mathbf{InL}| - \lambda$ bits of $\mathbf{OutL}_b[\mathsf{cnt}_b]$ — the last λ bits are reserved for garbling the finalization signal.
- $\{\!\!\{\mathsf{st}\}\!\!\}$ or $\{\!\!\{1_L, 1_R\}\!\!\} \leftarrow \mathsf{Finalize}(\mathsf{Gmem}, \{\!\!\{1\}\!\!\})$:
  - If $\mathbf{InitL} \neq \emptyset$, the output should be of the form $\{\!\!\{1_L, 1_R\}\!\!\}$, where $\mathsf{Lbl}(\{\!\!\{1_L\}\!\!\})$ is the last 2λ bits of $\mathbf{OutL}_0[\mathsf{cnt}_0]$ and $\mathsf{Lbl}(\{\!\!\{1_R\}\!\!\})$ is the last 2λ bits of $\mathbf{OutL}_1[\mathsf{cnt}_1]$. Here, we use the subscripts "L" and "R" are used to differentiate the two 1 bits;
  - Else if $\mathbf{InitL} = \emptyset$ the output should be of the form $\{\!\!\{\mathsf{st}\}\!\!\}$; furthermore, st is of the form $\mathsf{st} := (\mathsf{st}_0, \mathsf{st}_1)$, such that for $b \in \{0, 1\}$, each st_b is a bit vector containing exactly cnt_b and padded with 0s to a length of exactly B, where cnt_b is the total number of times Switch has been invoked a direction bit $\mathsf{leaf}[0] = b$; and moreover, $\mathsf{Lbl}(\{\!\!\{\mathsf{st}\}\!\!\}) = \mathbf{FinL}$.

Remark 3 (Two types of $\mathsf{GSwitch}$es depending on whether $\mathbf{InitL} = \emptyset$)**.** Later on in our construction (see also Figure 2), there will be two types of garbled switches, those that correspond to *empty* buckets of the Bucket ORAM (i.e., where $\mathbf{InitL} = \emptyset$) and those that correspond to *full* buckets (i.e., where $\mathbf{InitL} \neq \emptyset$). For the latter type ($\mathbf{InitL} \neq \emptyset$), when the switch first becomes active, its children switches have been operating for a while, and this is why we need to call its Init procedure to synchronize its state with its children. The input to the Init is passed down from the previous parent of their children, i.e., the garbled switch whose role it is taking over. The former type ($\mathbf{InitL} = \emptyset$) need not perform initialization, since their children switches are fresh when they first become active. When the latter type ($\mathbf{InitL} \neq \emptyset$) finalizes, its children need to be "rebuilt" as well; this is why it needs to pass the authenticated finalization signals $\{\!\!\{1_L, 1_R\}\!\!\}$ to its children.

5.3.2 Construction

Although Heath et al. [HKO21] describe a garbled switch scheme for constructing an access-revealing garbled one-time memory, again we need a new variant that supports 1) dynamic finalization; and 2) the XOR trick. We therefore describe a new variant supporting these features. Our construction is explained in Figure 4 which calls the following subroutine for creating the garbled stacks. Note that when $\mathbf{InitL} \neq \emptyset$, we are using the variant of GStack that does not have a $\mathsf{Finalize}$ call (see Section 5.2.4).

Subroutine for creating garbled stacks

- Let $m = 2\mathsf{B}$ and let $w = |\mathbf{OutL}_0[0]|$. Parse $\mathbf{InL} := (\mathbf{InitL}, \_, \_)$.
- If $\mathbf{InitL} \neq \emptyset$, then: let $t_{\max} = 4\mathsf{B}{+}1$; parse $\mathbf{InitL} := (\mathbf{InitL}_0, \mathbf{InitL}_1)$, and let $\mathbf{ctrlL} \overset{\$}{\leftarrow} \{0, 1\}^{\lambda}$.

Evaluator	**Garbler**
• $\mathsf{Init}(\{\!\{\mathsf{st}^E_\emptyset\}\!\})$: *// called when* $\mathbf{InitL} \neq \emptyset$	
1. parse $\{\!\{\mathsf{st}^E_\emptyset\}\!\} := \{\!\{\beta_{b,i}\}\!\}_{b\in\{0,1\},i\in[0:2\mathsf{B})}$; 2. for $b \in \{0,1\}$, $i \in [0 : 2\mathsf{B})$, call $\mathsf{GStack}_b.\mathsf{Pop}(\{\!\{\beta_{b,i}\}\!\})$;	Create two garbled stacks GStack_0 and GStack_1 as explained in a separate subroutine;
• $\mathsf{Switch}(\{\!\{\mathsf{leaf}^E\}\!\}, \{\!\{\mathsf{addr}\}\!\}, \llbracket L \rrbracket)$:	
1. Call $\{\!\{\beta_0 = \mathsf{leaf}[0]\}\!\}, \{\!\{\beta_1 = 1 - \beta_0\}\!\}, \{\!\{\mathsf{leaf}' = \mathsf{leaf}[1:]\}\!\}, \{\!\{\mathsf{addr}' = \mathsf{addr}\}\!\}, \llbracket L' = L \oplus \mathbf{RdL}[\tau] \rrbracket \leftarrow \mathsf{GCSw}_\tau(\{\!\{\mathsf{leaf}\}\!\}, \{\!\{\mathsf{addr}\}\!\}, \llbracket L \rrbracket, \llbracket \mathbf{RdL}[\tau] \rrbracket)$;	For each $\tau \in [0 : 2\mathsf{B})$, create the sharing $\llbracket \mathbf{RdL}[\tau] \rrbracket$, and garble the GCSw_τ circuit (whose functionality is defined on the left);
2. For $b \in \{0,1\}$, $\llbracket K_b, \_ \rrbracket \leftarrow \mathsf{GStack}_b.\mathsf{Pop}(\{\!\{\beta_b\}\!\})$; *// here _ means ignore the last* 2λ *bits* 3. Output $\left(\{\!\{\mathsf{leaf}'\}\!\}, \{\!\{\mathsf{addr}'\}\!\}, \llbracket L' \rrbracket\right) \oplus \llbracket K_0 \rrbracket \oplus \llbracket K_1 \rrbracket \oplus \mathsf{TrL}_\tau$.	For each $\tau \in [0 : 2\mathsf{B})$, compute the translation label $\mathsf{TrL}_\tau := \mathsf{Lbl}(\llbracket K_0 \rrbracket_\tau) \oplus \mathsf{Lbl}(\llbracket K_1 \rrbracket_\tau) \oplus \mathsf{Lbl}(\{\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\}_\tau, \llbracket L' \rrbracket_\tau)$;
• $\mathsf{Finalize}(\{\!\{1\}\!\})$:	
1. If $\mathbf{InitL} \neq \emptyset$, call: – $\{\!\{1_L, 1_R\}\!\} \leftarrow \mathsf{Dec}_{\{\!\{1\}\!\}}(\mathsf{ct}_\tau)$; – $\llbracket \_, K'_0 \rrbracket \leftarrow \mathsf{GStack}_0.\mathsf{Pop}(\{\!\{1_L\}\!\})$; – $\llbracket \_, K'_1 \rrbracket \leftarrow \mathsf{GStack}_1.\mathsf{Pop}(\{\!\{1_R\}\!\})$; – $\{\!\{1'_L\}\!\} := \{\!\{1\}\!\} \oplus \llbracket K'_0 \rrbracket \oplus \mathsf{TrL}'_{0,\tau}$; – $\{\!\{1'_R\}\!\} := \{\!\{1\}\!\} \oplus \llbracket K'_1 \rrbracket \oplus \mathsf{TrL}'_{1,\tau}$; and output $\{\!\{1'_L, 1'_R\}\!\}$.	If $\mathbf{InitL} \neq \emptyset$, then, for each $\tau \in [0 : 2\mathsf{B}]$, create the ciphertext $\mathsf{ct}_\tau = \mathsf{Enc}_{\{\!\{1\}\!\}_\tau}(\{\!\{1_L, 1_R\}\!\}_\tau)$, and compute $\mathsf{TrL}'_{0,\tau} := \mathsf{Lbl}(\llbracket K'_0 \rrbracket_\tau) \oplus \mathsf{Lbl}(\{\!\{1\}\!\}_\tau)$ and $\mathsf{TrL}'_{1,\tau} := \mathsf{Lbl}(\llbracket K'_1 \rrbracket_\tau) \oplus \mathsf{Lbl}(\{\!\{1\}\!\}_\tau)$.
2. Else, call – $\{\!\{\mathsf{st}_0\}\!\} \leftarrow \mathsf{GStack}_0.\mathsf{Finalize}(\{\!\{1\}\!\})$, – $\{\!\{\mathsf{st}_1\}\!\} \leftarrow \mathsf{GStack}_1.\mathsf{Finalize}(\{\!\{1\}\!\})$, and output $\{\!\{\mathsf{st}_0, \mathsf{st}_1\}\!\}$.	

Figure 4: GSwitch algorithm.

For $b \in \{0,1\}$, let $\mathbf{IL}_b = \mathbf{InitL}_b || \mathrm{rand}() || \mathbf{ctrlL} \in \{0,1\}^{m\cdot\lambda + 2t_{\max}\cdot\lambda}$; let $\mathbf{OL}_b \xleftarrow{\$} \{0,1\}^{t_{\max}\cdot(w+\lambda)}$.

- Else, let $t_{\max} = 2\mathsf{B}$; let $\mathbf{ctrlL} \xleftarrow{\$} \{0,1\}^{\lambda}$, for $b \in \{0,1\}$, let $\mathbf{IL}_b = \mathrm{rand}() || \mathbf{ctrlL} \in \{0,1\}^{2t_{\max}\cdot\lambda}$; let $\mathbf{OL}_b = \mathrm{rand}() || \mathbf{FinL} \in \{0,1\}^{t_{\max}\cdot w + m\cdot\lambda}$.
  // GStack_0 *and* GStack_1 *share the same finalization signal labels for all time steps*
- For $b \in \{0,1\}$: call $(\mathsf{GStack}_b.\mathsf{Gmem}, \mathsf{GStack}_b.\mathsf{GC}) \leftarrow \mathsf{GStack}_b.\mathsf{Garble}(1^{\lambda}, \mathsf{sk}, \mathsf{params} = (m, w, t_{\max}), \mathbf{DB} = \mathbf{OutL}_b, \mathbf{IL}_b, \mathbf{OL}_b)$.

In Figure 4, when we write the evaluator's algorithm, we do not explicitly write the time step τ, however, keep in mind that the inputs and outputs of Switch as well as the inputs to $\mathsf{Finalize}$ are actually encoded using τ-dependent labels. When we write the garbler's algorithm, since the garbler must create some garbled circuitry per time step τ, we explicitly write out the current time step τ in subscript, e.g., $\{\!\!\{\mathsf{var}\}\!\!\}_{\tau}$ means the variable garbled under a τ-dependent label.

It is not hard to see that the construction in Figure 4 costs $O(\mathsf{B}\cdot(\lambda\cdot w_1 + w_2)\cdot\log\mathsf{B})$ bits, where $w_1 = |\mathsf{leaf}| + |\mathsf{addr}|$ and $w_2 = |L|$.

5.3.3 Security Proof

We show that there exists a p.p.t. simulator Sim such that the simulated view is computationally indistinguishable from the real-world view. Since the syntax of $\mathsf{GSwitch}$ is slightly different from the standard syntax of a garbled data structure defined in Section 3.3, we note that in the security definition, we fix the $\mathbf{RdL}$, $\mathbf{OutL}_0$, $\mathbf{OutL}_1$, and $\mathbf{FinL}$ parts to be arbitrary strings. The labels $\mathbf{InL}[S]$ are generated at random, and the input labels $\mathbf{InL}[\neg S]$ are fixed to be an arbitrary string.

Below we describe the proof *for the case when all inputs are free inputs.* The case when some inputs are fixed inputs can be proven in a very similar manner.

Case 1: $\mathbf{InitL} = \emptyset$. We first describe the simulator construction.

- **Input:** t^*, the outputs $\{\!\!\{\mathsf{st}_0, \mathsf{st}_1\}\!\!\}$, $\{\{\!\!\{\mathsf{outp}\}\!\!\}_{\tau}\}_{\tau\in[0:t^*)}$ where $\mathsf{outp}_{\tau} = \left(\{\!\!\{\mathsf{leaf}'\}\!\!\}_{\tau}, \{\!\!\{\mathsf{addr}'\}\!\!\}_{\tau}, \llbracket L'_{\tau}\rrbracket\right) \oplus \llbracket K_0\rrbracket_{\tau} \oplus \llbracket K_1\rrbracket_{\tau} \oplus \mathsf{TrL}_{\tau}$, and the leakage $\{\mathsf{leaf}_{\tau}\}_{\tau\in[0:t^*)}$;
- **Output:** simulated garbled inputs $\{\!\!\{1\}\!\!\}_{t^*}$, $\{\{\!\!\{\mathsf{leaf}, \mathsf{addr}\}\!\!\}_{\tau}, \llbracket L\rrbracket_{\tau}\}_{\tau\in[0:t^*)}$, and simulated garbled circuitry and initial garbled memory including the following terms: $\{\mathsf{GCSw}_{\tau}.\mathsf{GC}, \llbracket \mathbf{RdL}[\tau]\rrbracket\}_{\tau\in[0:t_{\max})}$, $\{\mathsf{TrL}_{\tau}\}_{\tau\in[0:t_{\max})}$, and $\{\mathsf{GStack}_b.(\mathsf{GC}, \mathsf{Gmem})\}_{b\in\{0,1\}}$.
- **Construction:** The simulator proceeds as follows and outputs the underlined terms where $t_{\max} = 2\mathsf{B}$:
  - For $\tau \in [0 : t^*)$, choose $\{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_{\tau}, \llbracket L'\rrbracket_{\tau}$, $\llbracket K_0, K_1\rrbracket_{\tau}$, and $\underline{\mathsf{TrL}_{\tau}}$ at random subject to $(\{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_{\tau}, \llbracket L'\rrbracket_{\tau}) \oplus \llbracket K_0\rrbracket_{\tau} \oplus \llbracket K_1\rrbracket_{\tau} \oplus \mathsf{TrL}_{\tau} = \{\!\!\{\mathsf{outp}\}\!\!\}_{\tau}$. Moreover, for $\tau \in [0 : t^*)$, choose $\llbracket K'_0, K'_1\rrbracket_{\tau}$ at random. For $\tau \in [t^*, t_{\max})$, choose $\{\!\!\{\beta_0, \beta_1, \mathsf{leaf}', \mathsf{addr}'\}\!\!\}_{\tau}$ and $\llbracket L'\rrbracket_{\tau}$ at random, choose $\underline{\mathsf{TrL}_{\tau}}$ at random.
  - Call

$$\underline{\mathsf{GStack}_0.(\mathsf{GC}, \mathsf{Gmem}), \{\!\!\{1\}\!\!\}_{t^*}}, \{\!\!\{\beta_0\}\!\!\}_{\tau\in[0:t^*)}$$
$$\leftarrow \mathsf{GStack}_0.\mathsf{Sim}(1^{\lambda}, \mathsf{GStack}_0.\mathsf{params}, t^*, \{\llbracket K_0, K'_0\rrbracket_{\tau}\}_{\tau\in[0:t^*)}, \{\!\!\{\mathsf{st}_0\}\!\!\}, \{\mathsf{leaf}_{\tau}[0]\}_{\tau\in[0:t^*)});$$

  - Call

$$\underline{\mathsf{GStack}_1.(\mathsf{GC}, \mathsf{Gmem})}, \{\!\!\{\beta_1\}\!\!\}_{\tau\in[0:t^*)}$$
$$\leftarrow \mathsf{GStack}_1.\mathsf{Sim}(1^{\lambda}, \mathsf{GStack}_1.\mathsf{params}, t^*, \{\llbracket K_1, K'_1\rrbracket_{\tau}\}_{\tau\in[0:t^*)}, \{\!\!\{\mathsf{st}_1\}\!\!\}, \{\!\!\{1\}\!\!\}_{t^*}, \{\mathsf{leaf}_{\tau}[0]\}_{\tau\in[0:t^*)})$$

- For $\tau \in [0 : t^*)$,

  $$\underline{\mathsf{GCSw}_\tau.\mathsf{GC}, \llbracket \mathbf{RdL}[\tau] \rrbracket, \{\!\!\{\mathsf{leaf}, \mathsf{addr}\}\!\!\}_\tau, \llbracket L \rrbracket_\tau} \leftarrow \mathsf{GCSw}_\tau.\mathsf{Sim}(1^\lambda, \{\!\!\{\beta_0, \beta_1\}\!\!\}_\tau, \{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau, \llbracket L' \rrbracket_\tau);$$

  for $t \in [t^*, t_{\max})$, call $\underline{\mathsf{GCSw}_\tau.\mathsf{GC}, \llbracket \mathbf{RdL}[\tau] \rrbracket}, \_ \leftarrow \mathsf{GCSw}_\tau.\mathsf{Sim}(1^\lambda, \{\!\!\{\beta_0, \beta_1\}\!\!\}_\tau, \{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau, \llbracket L' \rrbracket_\tau)$.

To argue that the above simulated terms are computationally indistinguishable from the real-world counterparts, we can do a straightforward hybrid argument. We can start with the real world view which we denote as Hyb_0. Then, we will replace each garbled component with a simulated counterpart, and we shall do this in reverse topological order of the dependency graph.

- Hyb_1: otherwise identical as Hyb_0, except that now we call $\mathsf{GStack}_0.(\mathsf{GC}, \mathsf{Gmem})$, $\{\!\!\{1\}\!\!\}_{t^*}$, $\{\!\!\{\beta_0\}\!\!\}_{\tau \in [0:t^*)}$ $\leftarrow \mathsf{GStack}_0.\mathsf{Sim}(1^\lambda$, $\mathsf{GStack}_0.\mathsf{params}$, t^*, $\{\llbracket K_0, K'_0 \rrbracket_\tau\}_{\tau \in [0:t^*)}$, $\{\!\!\{\mathsf{st}_0\}\!\!\}$, $\{\mathsf{leaf}_\tau[0]\}_{\tau \in [0:t^*)})$ to generate the terms $\mathsf{GStack}_0.(\mathsf{GC}, \mathsf{Gmem})$, $\{\!\!\{1\}\!\!\}_{t^*}$, $\{\!\!\{\beta_0\}\!\!\}_{\tau \in [0:t^*)}$.
- Hyb_2: otherwise identical as Hyb_1, except that now we call $\mathsf{GStack}_1.(\mathsf{GC}, \mathsf{Gmem})$, $\{\!\!\{\beta_1\}\!\!\}_{\tau \in [0:t^*)}$ $\leftarrow \mathsf{GStack}_1.\mathsf{Sim}(1^\lambda$, $\mathsf{GStack}_0.\mathsf{params}$, t^*, $\{\llbracket K_1, K'_1 \rrbracket_\tau\}_{\tau \in [0:t^*)}$, $\{\!\!\{\mathsf{st}_1\}\!\!\}$, $\{\!\!\{1\}\!\!\}_{t^*}$, $\{\mathsf{leaf}_\tau[0]\}_{\tau \in [0:t^*)})$ to generate the terms $\mathsf{GStack}_1.(\mathsf{GC}, \mathsf{Gmem})$, $\{\!\!\{\beta_1\}\!\!\}_{\tau \in [0:t^*)}$.
- Hyb_3: otherwise identical as Hyb_2 except that now, for $\tau \in [0 : t^*)$, we choose $\{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau$, $\llbracket L' \rrbracket_\tau$, and TrL_τ at random subject to $(\{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau, \llbracket L' \rrbracket_\tau) \oplus \llbracket K_0 \rrbracket_\tau \oplus \llbracket K_1 \rrbracket_\tau \oplus \mathsf{TrL}_\tau = \{\!\!\{\mathsf{outp}_\tau\}\!\!\}$. For $\tau \in [t^*, t_{\max})$, we choose the active encodings $\{\!\!\{\beta_0, \beta_1, \mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau$ and $\llbracket L' \rrbracket_\tau$ at random. Note that the labels of $\{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau$, $\llbracket L' \rrbracket_\tau$ are needed in the garbling of GCSw_τ.
- Hyb_4: otherwise identical as Hyb_3, except that now for $\tau \in [0 : t_{\max})$, we call $\mathsf{GCSw}_\tau.\mathsf{GC}$, $\llbracket \mathbf{RdL}[\tau] \rrbracket$, $\{\!\!\{\mathsf{leaf}, \mathsf{addr}\}\!\!\}_\tau, \llbracket L \rrbracket_\tau \leftarrow \mathsf{GCSw}_\tau.\mathsf{Sim}(1^\lambda$, $\{\!\!\{\beta_0, \beta_1\}\!\!\}_\tau$, $\{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau$, $\llbracket L' \rrbracket_\tau)$ to generate the terms $\mathsf{GCSw}_\tau.\mathsf{GC}$, $\llbracket \mathbf{RdL}[\tau] \rrbracket$, $\{\!\!\{\mathsf{leaf}, \mathsf{addr}\}\!\!\}_\tau$, and $\llbracket L \rrbracket_\tau$. Note that for $\tau \in [t^*, t_{\max})$, we only need to include $\mathsf{GCSw}_\tau.\mathsf{GC}$ and $\llbracket \mathbf{RdL}[\tau] \rrbracket$ in the output, and the terms $\{\!\!\{\mathsf{leaf}, \mathsf{addr}\}\!\!\}_\tau$, and $\llbracket L \rrbracket_\tau$ are thrown away.

Due to the security of GStack, Hyb_1 is computationally indistinguishable from Hyb_0, and Hyb_2 is computationally indistinguishable from Hyb_1. Hyb_3 is identically distributed as Hyb_2. Due to the security of garbled circuits, Hyb_4 and Hyb_3 are computationally indistinguishable. Finally, observe that Hyb_4 is the same as our simulator construction.

Case 2: $\mathbf{InitL} \neq \emptyset$. We first describe the simulator construction.

- **Input:** t^*, the outputs $\{\!\!\{1'_L, 1'_R\}\!\!\}$, $\{\{\!\!\{\mathsf{outp}\}\!\!\}_\tau\}_{\tau \in [0:t^*)}$ where $\mathsf{outp}_\tau = \big(\{\!\!\{\mathsf{leaf}'\}\!\!\}_\tau, \{\!\!\{\mathsf{addr}'\}\!\!\}_\tau, \llbracket L'_\tau \rrbracket\big) \oplus \llbracket K_0 \rrbracket_\tau \oplus \llbracket K_1 \rrbracket_\tau \oplus \mathsf{TrL}_\tau$, and the leakage $\{\mathsf{leaf}_\tau\}_{\tau \in [0:t^*)}$ and $\mathsf{st}_\emptyset$;
- **Output:** simulated garbled inputs $\{\!\!\{1\}\!\!\}_{t^*}$, $\{\{\!\!\{\mathsf{leaf}, \mathsf{addr}\}\!\!\}_\tau, \llbracket L \rrbracket_\tau\}_{\tau \in [0:t^*)}$, $\{\!\!\{\mathsf{st}_\emptyset\}\!\!\}$ and simulated garbled circuitry and initial garbled memory including the following terms: $\{\mathsf{GCSw}_\tau.\mathsf{GC}, \llbracket \mathbf{RdL}[\tau] \rrbracket\}_{\tau \in [0:t_{\max})}$, $\{\mathsf{TrL}_\tau\}_{\tau \in [0:t_{\max})}$, $\{\mathsf{TrL}'_{0,\tau}, \mathsf{TrL}'_{1,\tau}\}_{\tau \in [0:t_{\max}]}$, $\{\mathsf{ct}_\tau\}_{\tau \in [0:t_{\max}]}$, and $\{\mathsf{GStack}_b.(\mathsf{GC}, \mathsf{Gmem})\}_{b \in \{0,1\}}$.
- **Construction:** The simulator proceeds as follows and outputs the underlined terms — below, we sometimes use $\{\!\!\{1_0\}\!\!\}, \{\!\!\{1'_0\}\!\!\}$ as aliases for $\{\!\!\{1_L\}\!\!\}$ and $\{\!\!\{1'_L\}\!\!\}$ respectively, and use $\{\!\!\{1_1\}\!\!\}$, $\{\!\!\{1'_1\}\!\!\}$ as aliases for $\{\!\!\{1_R\}\!\!\}$ and $\{\!\!\{1'_R\}\!\!\}$, respectively;
  - For $\tau \in [0 : t^*)$, choose $\{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau, \llbracket L' \rrbracket_\tau$, $\llbracket K_0, K_1 \rrbracket_\tau$, and $\underline{\mathsf{TrL}_\tau}$ at random subject to $(\{\!\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau, \llbracket L' \rrbracket_\tau) \oplus \llbracket K_0 \rrbracket_\tau \oplus \llbracket K_1 \rrbracket_\tau \oplus \mathsf{TrL}_\tau = \{\!\!\{\mathsf{outp}\}\!\!\}_\tau$. Moreover, choose $\llbracket K_0, K_1 \rrbracket_{t^*}$ at random; choose $\{\llbracket K'_0, K'_1 \rrbracket_\tau\}_{\tau \in [0:t^*]}$ at random. For $\tau \in [t^*, t_{\max})$, choose $\underline{\mathsf{TrL}_\tau}$ at random, and choose the terms $\{\!\!\{\beta_0, \beta_1, \mathsf{leaf}', \mathsf{addr}'\}\!\!\}_\tau, \llbracket L' \rrbracket_\tau$ at random;

- For $b \in \{0,1\}$, choose $\llbracket Z_{b,0}, \ldots, Z_{b,2\mathsf{B}-1} \rrbracket$ at random to represent the outputs of $\mathsf{GStack}_b.\mathsf{Pop}$ inside Init;
- For $b \in \{0,1\}$, call

  $$\underline{\mathsf{GStack}_b.(\mathsf{GC}, \mathsf{Gmem})}, \underline{\{\!\{1_b\}\!\}_{t^*}}, \underline{\{\!\{\beta_b\}\!\}_{\tau \in [0:t^*)}}, \underline{\{\!\{\mathsf{st}_\emptyset\}\!\}}$$
  $$\leftarrow \mathsf{GStack}_b.\mathsf{Sim}(1^\lambda, \mathsf{GStack}_b.\mathsf{params}, t^*, \llbracket Z_{b,0}, \ldots, Z_{b,2\mathsf{B}-1} \rrbracket, \{\llbracket K_b, K'_b \rrbracket_\tau\}_{\tau \in [0:t^*]}, \mathsf{st}_\emptyset, \{\mathsf{leaf}_\tau[0]\}_{\tau \in [0:t^*)}).$$

- For $b \in \{0,1\}$, for $\tau \in [0 : t^*]$, choose $\underline{\mathsf{TrL}'_{b,\tau}} = \llbracket K'_b \rrbracket_\tau \oplus \{\!\{1_b\}\!\}_\tau \oplus \{\!\{1'_b\}\!\}$, and for $t \in [t^* : t_{\max}]$, choose $\underline{\mathsf{TrL}'_{b,\tau}}$ at random;
- For $\tau \in [0 : t_{\max}]$ and $\tau \neq t^*$, let $\underline{\mathsf{ct}_\tau}$ be encryptions of 0 under random keys. Pick some random $\underline{\{\!\{1\}\!\}_{t^*}}$, and let $\underline{\mathsf{ct}_{t^*}}$ be an honest encryption of $\{\!\{1_L, 1_R\}\!\}_{t^*}$ using the key $\{\!\{1\}\!\}_{t^*}$;
- For $\tau \in [0 : t^*)$,

  $$\underline{\mathsf{GCSw}_\tau.\mathsf{GC}, \llbracket \mathbf{RdL}[\tau] \rrbracket, \{\!\{\mathsf{leaf}, \mathsf{addr}\}\!\}_\tau, \llbracket L \rrbracket_\tau} \leftarrow \mathsf{GCSw}_\tau.\mathsf{Sim}(1^\lambda, \{\!\{\beta_0, \beta_1\}\!\}_\tau, \{\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\}_\tau, \llbracket L' \rrbracket_\tau);$$

  for $t \in [t^*, t_{\max})$, call $\underline{\mathsf{GCSw}_\tau.\mathsf{GC}, \llbracket \mathbf{RdL}[\tau] \rrbracket}, \_ \leftarrow \mathsf{GCSw}_\tau.\mathsf{Sim}(1^\lambda, \{\!\{\beta_0, \beta_1\}\!\}_\tau, \{\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\}_\tau, \llbracket L' \rrbracket_\tau)$.

We now show that these simulated terms are computationally indistinguishable from their real-world counterparts. To prove this, we will go through a sequence of hybrid experiments, starting from the real-world view, and in a reverse topological order of the dependency graph, swap each garbled component with a simulated one. Henceforth, let Hyb_0 denote the real-world view.

- Hyb_1: otherwise identical as Hyb_0, except that now, for $b \in \{0,1\}$, we call $\mathsf{GStack}_b.(\mathsf{GC}, \mathsf{Gmem})$, $\{\!\{1_b\}\!\}_{t^*}$, $\{\!\{\beta_b\}\!\}_{\tau \in [0:t^*)}$, $\{\!\{\mathsf{st}_\emptyset\}\!\} \leftarrow \mathsf{GStack}_b.\mathsf{Sim}(1^\lambda, \mathsf{GStack}_b.\mathsf{params}, t^*, \llbracket Z_{b,0}, \ldots, Z_{b,2\mathsf{B}-1} \rrbracket, \{\llbracket K_b, K'_b \rrbracket_\tau\}_{\tau \in [0:t^*]}$, $\mathsf{st}_\emptyset$, $\{\mathsf{leaf}_\tau[0]\}_{\tau \in [0:t^*)})$ to generate the terms $\mathsf{GStack}_b.(\mathsf{GC}, \mathsf{Gmem})$, $\{\!\{1_b\}\!\}_{t^*}$, $\{\!\{\beta_b\}\!\}_{\tau \in [0:t^*)}$, $\{\!\{\mathsf{st}_\emptyset\}\!\}$.
- Hyb_2: otherwise identical as Hyb_1 except that now choose $\{\mathsf{TrL}_\tau\}_{\tau \in [0:t^*)}$, and $\{\mathsf{TrL}'_{0,\tau}, \mathsf{TrL}'_{1,\tau}\}_{\tau \in [0:t^*]}$ subject to $(\{\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\}_\tau, \llbracket L' \rrbracket_\tau) \oplus \llbracket K_0 \rrbracket_\tau \oplus \llbracket K_1 \rrbracket_\tau \oplus \mathsf{TrL}_\tau = \mathsf{outp}_\tau$, and $\mathsf{TrL}'_{b,\tau} = \llbracket K'_b \rrbracket_\tau \oplus \{\!\{1_b\}\!\}_\tau \oplus \{\!\{1'_b\}\!\}$. For $\tau \in [t^*, t_{\max})$, we choose TrL_τ at random, and for $\tau \in [t^*, t_{\max}]$, for $b \in \{0,1\}$, we choose $\mathsf{TrL}'_{b,\tau}$ at random.
- Hyb_3: otherwise identical to Hyb_2 except that now for $\tau \in [0 : t_{\max}]$ and $\tau \neq t^*$, we let ct_τ be an encryption of 0 under a randomly chosen key; additionally, we choose $\{\!\{1\}\!\}_{t^*}$ at random, and let ct_{t^*} be an honest encryption of $\{\!\{1_L, 1_R\}\!\}$ under the key $\{\!\{1\}\!\}_{t^*}$.
- Hyb_4: otherwise identical to Hyb_3, except that now for $\tau \in [0 : t_{\max})$, we call $\mathsf{GCSw}_\tau.\mathsf{GC}$, $\llbracket \mathbf{RdL}[\tau] \rrbracket$, $\{\!\{\mathsf{leaf}, \mathsf{addr}\}\!\}_\tau$, $\llbracket L \rrbracket_\tau \leftarrow \mathsf{GCSw}_\tau.\mathsf{Sim}(1^\lambda$, $\{\!\{\beta_0, \beta_1\}\!\}_\tau$, $\{\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\}_\tau, \llbracket L' \rrbracket_\tau)$ to generate the terms $\mathsf{GCSw}_\tau.\mathsf{GC}$, $\llbracket \mathbf{RdL}[\tau] \rrbracket$, $\{\!\{\mathsf{leaf}, \mathsf{addr}\}\!\}_\tau$, $\llbracket L \rrbracket_\tau$, where for $\tau \in [t^*, t_{\max})$, the terms $\{\!\{\beta_0, \beta_1\}\!\}_\tau$, $\{\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\}_\tau, \llbracket L' \rrbracket_\tau)$ are generated at random.

Hyb_1 is computationally indistinguishable from Hyb_0 due to the security of GStack, Hyb_2 is identically distributed as Hyb_1, Hyb_3 and Hyb_2 are computationally indistinguishable due to the semantic security of the encryption scheme, and Hyb_4 and Hyb_3 are also computationally indistinguishable due to the security of garbled circuit. Finally, observe that Hyb_4 is the same as our simulator construction.

5.4 Leaf Switch (GLeafSwitch)

Leaf switches GLeafSwitch will later be used by leaf nodes in the ORAM tree. Leaf switches need not perform any routing, it only outputs a translation label TrL. The syntax of GLeafSwitch is as

Evaluator	**Garbler**
- $\mathsf{Switch}([\![L]\!])$: call $L \oplus \mathbf{RdL}[\tau] \leftarrow \mathsf{GC}_\tau([\![L]\!], [\![\mathbf{RdL}[\tau]]\!])$	for $\tau \in [0 : t_{\max})$: create the garbled state $[\![\mathbf{RdL}[\tau]]\!]$, and the garbled circuit GC_τ.

Figure 5: GLeafSwitch algorithm

follows. We assume that $\mathsf{params} = (m = t_{\max}, w)$ where $m = t_{\max}$ denotes the maximum number of times the switch can be invoked, and it is also the length of the initial array of labels $\mathbf{RdL}$; w denotes the bit-width of each entry in $\mathbf{RdL}$ as well as the bit-width of L.

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{RdL})$: upon receiving an array of labels $\mathbf{RdL}$, output Gmem and GC;
- $\mathsf{TrL} \leftarrow \mathsf{Switch}^{\mathsf{GC}}(\mathsf{Gmem}, [\![L]\!])$: correctness requires that $\mathsf{TrL} = \mathbf{RdL}[\tau] \oplus L$ where τ is the current local time.

Construction. The construction of a leaf switch is standard and described in Figure 5. The cost of our GLeafSwitch construction is $O(t_{\max} \cdot w)$ bits, using the efficient operations on shared values mentioned in Section 3.2. Its security proof also follows directly from the security of garbled circuits.

5.5 Garbled Access-Revealing One-Time Memory (GOTM)

5.5.1 Definition

A garbled access-revealing one-time memory implements a memory array such that once initialized, it is promised that each real address will only be accessed once; further, the evaluator knows exactly which address is being accessed. Let $\mathsf{params} = (m = t_{\max}, w)$ where $m = t_{\max}$ denotes the number of elements in the initial data array $\mathbf{DB}$, as well as the maximum number of requests, and w is the bit-width of each element. We allow two types of requests, *real* requests ask for a real logical address, and *filler* requests (denoted $(\perp, 0)$) do not ask for anything but they must be there for security. As mentioned, it is promised that *each real address will be requested at most once.*

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{InL}, \mathbf{OutL})$: receives input/output labels denoted $\mathbf{InL}$ and $\mathbf{OutL}$, and outputs $\mathsf{GC}, \mathsf{Gmem}$. We often write $\mathbf{InL} := (\mathbf{InitL}, \mathbf{ReqL}, \mathbf{CtrlL})$ where the first part corresponds to the labels consumed by the Init procedure, the second part corresponds to the labels consumed by up to $t_{\max}$ calls to Read, and the third part corresponds to the labels consumed by a possible call to Finalize in any of the $t_{\max}$ time steps;
- $\mathsf{Gmem}' \leftarrow \mathsf{Init}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathbf{DB}\}\!\})$: called once upfront before any invocation of Read, initialize the garbled data structure with the initial memory array $\mathbf{DB}$;
- $\mathsf{Gmem}', \{\!\{\mathsf{res}\}\!\} \leftarrow \mathsf{Read}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{idx}^E\}\!\})$: it should be that $\mathsf{res} = \mathbf{DB}[\mathsf{idx}]$; moreover, $\mathsf{Lbl}(\{\!\{\mathsf{res}\}\!\})$ should be the corresponding τ-dependent output label contained in $\mathbf{OutL}$;
- $\{\!\{\mathbf{DB}'\}\!\} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{1\}\!\})$: upon receiving some finalization signal $\{\!\{1\}\!\}$ garbled with a τ-dependent label, output a garbled list of elements that have not been read. Concretely, for $i \in [0 : m)$, $\mathbf{DB}'[i] = \mathbf{DB}[i]$ if i has not been read; else $\mathbf{DB}'[i] = (\perp, 0)$. Further, $\mathsf{Lbl}(\{\!\{\mathbf{DB}'\}\!\})$ should match the fixed finalization labels contained in $\mathbf{OutL}$.

Heath et al. [HKO21] propose an efficient garbled implementation of an access-revealing one-time memory. We modify their construction to support an additional $\mathsf{Finalize}$ procedure.

5.5.2 Construction

We can construct such a garbled, access-revealing one-time memory using GStack and a variant of $\mathsf{GSwitch}$. In Section 5.3, a non-leaf $\mathsf{GSwitch}$ has two types differentiated by whether $\mathbf{InitL} = \emptyset$ at the time of garbling. In this variant, there is no Init procedure or $\mathbf{InitL}$ label; moreover, $\mathbf{RdL} = \mathbf{0}$, and the input addr to the Switch function is replaced with a signal $\{\!\{1\}\!\}$. Finally, $\mathsf{Finalize}$ always outputs two control signals $\{\!\{1_L, 1_R\}\!\}$ garbled under a label popped out of the two stacks $\mathbf{OutL}_0$ and $\mathbf{OutL}_1$, respectively, to be passed to both the left and right children. More specifically, this variant of $\mathsf{GSwitch}$ has the following slightly simplified syntax: let $\mathsf{params} = (m = t_{\max}, \mathbf{w})$ where $t_{\max} = m$ denotes the maximum number of times the switch can be invoked (assumed to be a power of 2); and $\mathbf{w}$ contains the bit-widths of leaf and L which are inputs to the Switch procedure. $\mathbf{InL}$ contains m entries each of $\lambda \cdot (|\mathsf{leaf}| + 1) + |L|$ bits. For $b \in \{0, 1\}$, $\mathbf{OutL}_b$ contains $m/2 + 1$ entries each of $|L| + \lambda$ bits long. Each entry $\mathbf{OutL}_b[\tau]$ contains two parts: the first $|L|$ bits are the input label expected by the τ-th invocation of b-th child, and the last λ bits are the finalization label expected by the b-th child during its local time step τ. It is promised that among the m number of Switch requests, at most $m/2$ of them want to go left and at most $m/2$ of them want to go right.

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{InL}, \mathbf{OutL}_0, \mathbf{OutL}_1)$: creates garbled circuitry GC and initial garbled Gmem upon receiving the input labels $\mathbf{InL}$, and two stacks of output labels $\mathbf{OutL}_0$ and $\mathbf{OutL}_1$;
- $\mathsf{Gmem}', \{\!\{\mathsf{leaf}'\}\!\}, \{\!\{1'\}\!\}, [\![L']\!] \leftarrow \mathsf{Switch}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{leaf}^E\}\!\}, \{\!\{1\}\!\}, [\![L]\!])$: for correctness, the output satisfies $\mathsf{leaf}' = \mathsf{leaf}[1:]$, and $L' = L$; further, let $b = \mathsf{leaf}[0]$, and cnt_b be the number of times Switch has been invoked with the direction bit $\mathsf{leaf}[0] = b$ (not counting the current one). Then, it must be that $\mathsf{Lbl}(\{\!\{\mathsf{leaf}'\}\!\}, [\![L']\!])$ is the first $|L|$ bits of $\mathbf{OutL}_b[\mathsf{cnt}_b]$;
- $\{\!\{1_L, 1_R\}\!\} \leftarrow \mathsf{Finalize}(\mathsf{Gmem}, \{\!\{1\}\!\})$: outputs two garbled control signals where $\mathsf{Lbl}(\{\!\{1_L\}\!\})$ is the last λ bits of $\mathbf{OutL}_0[\mathsf{cnt}_0]$ and $\mathsf{Lbl}(\{\!\{1_R\}\!\})$ is the last λ bits of $\mathbf{OutL}_1[\mathsf{cnt}_1]$.

The construction of this slight variant of $\mathsf{GSwitch}$ is very similar to Figure 4. The only changes are 1) the parameters; 2) no Init call; and 3) in $\mathsf{Finalize}$, always take the branch that outputs $\{\!\{1_L, 1_R\}\!\}$. Such a $\mathsf{GSwitch}$ costs $O(m \cdot (\lambda w_1 + w_2) \cdot \log m)$ where $w_1 = |\mathsf{leaf}| + 1$ and $w_2 = |L|$.

Data structures. We will rely on a tree of $\mathsf{GSwitch}$es (or $\mathsf{GLeafSwitch}$es at the leaf level), and a $\mathsf{GSwitch}$ (or $\mathsf{GLeafSwitch}$) at level $\ell \in [0 : \ell_{\max}]$ of the tree is provisioned with a capacity of $2^{\ell_{\max} - \ell}$, where $\ell_{\max} = \log_2 m$. This means that the root (where $\ell = 0$) is provisioned to serve up to m route requests, each level 1 node must serve up to $m/2$ route requests, and so on. The leaf level will use a leaf-type switch denoted $\mathsf{GLeafSwitch}$, provisioned to serve at most 1 request.

Each leaf node not only has a $\mathsf{GLeafSwitch}$ instance, it also has a $\mathsf{GOTStack}$ instance associated with it — we also say that each $\mathsf{GOTStack}$ is *paired* with some $\mathsf{GLeafSwitch}$ if the two belong to the same leaf node. Each leaf represents a memory location, and the leaf's $\mathsf{GOTStack}$ stores the memory word at this location.

For levels $[0 : \ell_{\max} - 2]$, each $\mathsf{GSwitch}$'s two output label stacks $\mathbf{OutL}_0$ and $\mathbf{OutL}_1$ are filled with labels that match the two children's input and finalization labels over all time steps. In other words, if $\mathsf{GSwitch}$ is the parent of non-leaf switches $\mathsf{GSwitch}_0$ and $\mathsf{GSwitch}_1$, then for each $\tau \in [0 : \mathsf{GSwitch}.m/2]$, $\mathsf{GSwitch}.\mathbf{OutL}_b[\tau] = \mathsf{GSwitch}_b.\mathbf{InL}_b[\tau] || \mathsf{GSwitch}_b.\mathbf{CtrlL}_b[\tau]$. If $\mathsf{GSwitch}$ is the parent of two leaf switches denoted $\mathsf{GLeafSwitch}_0$ $\mathsf{GLeafSwitch}_1$, let $\mathsf{GOTStack}_0$ and $\mathsf{GOTStack}_1$ be the two

Evaluator

```
• $\mathsf{Init}(\{\!\{\mathbf{DB}\}\!\})$:
1. for $i \in [0:m)$: $\mathsf{GOTStack}_i.\mathsf{Push}(\{\!\{\mathbf{DB}[i]\}\!\})$;
• $\mathsf{Read}(\{\!\{\mathsf{idx}^E\}\!\})$:
1. let $\llbracket L^0 \rrbracket = \llbracket \mathbf{OutL}[\tau] \rrbracket$;
2. let $\{\!\{\mathsf{leaf}^0\}\!\} = \{\!\{\mathsf{idx}\}\!\}$, $\{\!\{1^0\}\!\} = \{\!\{1\}\!\}_\tau$;
3. let $V = \text{root}$ and $\ell_{\max} = \log_2 m$;
4. for $\ell \in [0:\ell_{\max})$:
   – $\{\!\{\mathsf{leaf}^{\ell+1}\}\!\}, \{\!\{1^{\ell+1}\}\!\}, \llbracket L^{\ell+1} \rrbracket$
         $\leftarrow \mathsf{GSwitch}^V.\mathsf{Switch}(\{\!\{\mathsf{leaf}^\ell\}\!\}, \{\!\{1^\ell\}\!\}, \llbracket L^\ell \rrbracket)$
   – $V \leftarrow V.\mathsf{child}[\mathsf{idx}[\ell]]$;
     // $V$ is now the $\mathsf{idx}$-th leaf
5. $\mathsf{TrL} \leftarrow \mathsf{GLeafSwitch}_{\mathsf{idx}}.\mathsf{Switch}(\llbracket L^{\ell_{\max}} \rrbracket)$;
6. $\{\!\{\mathsf{rdata}\}\!\} \leftarrow \mathsf{GOTStack}_{\mathsf{idx}}.\mathsf{Pop}(\{\!\{1^{\ell_{\max}}\}\!\})$;
7. output $\{\!\{\mathsf{rdata}\}\!\} \oplus \mathsf{TrL}$;
• $\mathsf{Finalize}(\{\!\{1\}\!\})$:
1. $\mathsf{RecFinalize}(\text{root}, \{\!\{1\}\!\})$ defined below;
2. output $\{\!\{\mathbf{DB}'\}\!\} = \{\{\!\{D_i\}\!\}\}_{i \in [0:m)}$.
   Subroutine RecFinalize($V, \{\!\{1\}\!\}$)
a) if $V$ is a leaf, let $i$ be its index:
      $\{\!\{D_i\}\!\} \leftarrow \mathsf{GOTStack}_i.\mathsf{Finalize}(\{\!\{1\}\!\})$
   and return; else continue with the following;
b) $\{\!\{1_L, 1_R\}\!\} \leftarrow \mathsf{GSwitch}^V.\mathsf{Finalize}(\{\!\{1\}\!\})$
c) let $U_L$ and $U_R$ be $V$'s left and right children;
d) $\mathsf{RecFinalize}(U_L, \{\!\{1_L\}\!\})$; $\mathsf{RecFinalize}(U_R, \{\!\{1_R\}\!\})$.
```

Garbler

```
for $i \in [0:m)$:                       // see narrative
    call $\mathsf{GOTStack}_i.\mathsf{Garble}$;

for $\tau \in [0:m)$:
    create the sharing $\llbracket \mathbf{OutL}[\tau] \rrbracket$;
    create the garbling $\{\!\{1\}\!\}_\tau$;

for each tree node $V$:                  // see narrative
    call $\mathsf{GSwitch}^V.\mathsf{Garble}$;

for each $i \in [0:m)$:                  // see narrative
    call $\mathsf{GLeafSwitch}_i.\mathsf{Garble}$.
```

Figure 6: GOTM algorithm.

stacks paired with these leaf switches, $\mathsf{GSwitch}.\mathbf{OutL}_b[0] = \mathsf{GLeafSwitch}.\mathbf{InL}_b[0]||\mathsf{GOTStack}_b.F$. The output label of $\mathsf{GOTStack}$ matches the $\mathbf{RdL}$ label for the paired $\mathsf{GLeafSwitch}$.

Algorithm. We describe our GOTM construction in Figure 6, where we use $\mathsf{GSwitch}^V$ to denote the $\mathsf{GSwitch}$ instance associated with tree node V, and we use $\mathsf{GLeafSwitch}_i$ and $\mathsf{GOTStack}_i$ to denote the $\mathsf{GLeafSwitch}$ or $\mathsf{GOTStack}$ instance at the i-th leaf node.

Intuitively, to access a memory location denoted by idx, the root of the tree receives a shared label $[\![L]\!]$. This shared label is routed over a path in the tree towards the leaf node indexed by idx, and at each level, the parent will translate the language $\mathsf{Lbl}([\![L]\!])$ to an appropriate language that depends on the child's local time. In the end, the corresponding leaf node will receive $[\![L]\!]$ shared using a language it can recognize, and it can output the memory word garbled under the label L.

When the GOTM is finalized, it must output the residual memory array garbled under some fixed labels. In the residual array, every location that has been visited must be replaced with a filler element $(\perp, 0)$. One challenge here is that at finalization time, some leaves have been visited and they are already in local time 1; whereas others have not been visited and are in local time 0. To resolve this issue, we have the tree pass down a garbled finalization signal $\{\!\{1\}\!\}$ from the root to *all* leaves, and in each step, the parent node will translate the finalization signal $\{\!\{1\}\!\}$ into the local-time-dependent languages of its children. At the very end, all $\mathsf{GOTStack}$s at the leaf level will receive a loal-time-dependent finalization signal $\{\!\{1\}\!\}$. If he $\mathsf{GOTStack}$ is still in local time 0, it will be able to fast forward its local clock to 1 by performing a filler Pop operation. When all $\mathsf{GOTStack}$s are in local time 1, they can all output the top of their respective stacks garbled under a fixed label.

Our GOTM construction costs $O\left(m \cdot \lambda \cdot (w + \log m) \cdot \log^2 m\right)$ bits.

5.5.3 Security Proof

The security proof is again rather mechanical. We will swap each garbled component one by one — including the garbled circuitry, garbled memory, and (part of) its garbled inputs — with a simulated counterpart. The crux is to perform the swaps in a reverse topological order of the dependency graph of these components. Let Hyb_0 be the real-world view. We now define the following sequence of hybrids:

- Hyb_1: otherwise identical to Hyb_0 except the following modification. Let $\{\!\{\mathsf{res}\}\!\}_\tau$ be the output of the τ-th Read operation. For $\tau \in [0 : t^*)$, select TrL_τ and $\{\!\{\mathsf{rdata}\}\!\}_\tau$ at random subject to $\mathsf{TrL}_\tau \oplus \{\!\{\mathsf{rdata}\}\!\}_\tau = \{\!\{\mathsf{res}\}\!\}_\tau$ — note that this implicitly determines the labels for TrL_τ and $\{\!\{\mathsf{rdata}\}\!\}_\tau$. Hyb_1 is identically distributed as Hyb_0.
- Hyb_2: otherwise identical to Hyb_1 except the following modification. For every $i \in [0 : m)$, replace $\mathsf{GOTStack}_i.\mathsf{GC}$, $\mathsf{GOTStack}_i.\mathsf{Gmem}$, its garbled inputs $\{\!\{\mathbf{DB}[i]\}\!\}$, garbled Pop signal, as well as its garbled finalization signal with the output of $\mathsf{GOTStack}_i.\mathsf{Sim}$. Hyb_2 is computationally indistinguishable from Hyb_1 due to the security of $\mathsf{GOTStack}$.
- Hyb_3: otherwise identical to Hyb_2 except the following modification. For every $i \in [0 : m)$, replace $\mathsf{GLeafSwitch}_i.\mathsf{GC}$, $\mathsf{GLeafSwitch}_i.\mathsf{Gmem}$, and its input $[\![L]\!]$ with the output of $\mathsf{GLeafSwitch}_i.\mathsf{Sim}$. Hyb_3 is computationally indistinguishable from Hyb_2 due to the security of $\mathsf{GLeafSwitch}$.
- $\mathsf{Hyb}_*^{\ell_{\max}-1}, \ldots, \mathsf{Hyb}_*^0$: each of these hybrid experiments Hyb_*^ℓ is otherwise identical to the previous one, except that we now swap all $\mathsf{GSwitch}$ instances at level ℓ of the tree (including its garbled circuitry, garbled memory, as well as its garbled inputs) with the outcome of $\mathsf{GSwitch}.\mathsf{Sim}$. To prove the more generalized notion of simulation (Definition 1), for the $\mathsf{GSwitch}$ on the first level,

part of its garbled inputs might already be fixed but this does not matter if the underlying components satisfy the generalized notion of simulation, too. Each hybrid is computationally indistinguishable from the previous one, due to the security of $\mathsf{GSwitch}$. Note that to make these swaps, we need to make use of the leakage, which contains the index idx of each Read request.

5.6 Bucket (GBkt)

A GBkt implements an access-hiding one-time dictionary. We do not directly use the existing GDict instantiation (Section 4.2) for two reasons: 1) we need a $\mathsf{Finalize}$ function that is not supported by existing schemes; 2) our construction in this section treats metadata and data blocks separately, and thus we improve the asymptotical performance when block size is large. The garbled data structure should implement the following functionality. The Init function should be called once upfront. Init receives an initial memory array $\mathbf{DB}$. Each element of $\mathbf{DB}$ is either a *real* element of the format $(\mathsf{addr}, \mathsf{val})$, where $\mathsf{addr} \neq \bot$ is the logical address of the element, and val denotes its contents, or a *filler* element denoted $(\mathsf{addr} = \bot, \mathsf{val} = 0)$. The space of the addr can be larger than the capacity of the GBkt. Each Read operation asks for an element at a specified addr, and if such an element is found, its contents are returned; else, 0 is returned. When $\mathsf{Finalize}$ is called, it returns an $\mathbf{DB}'$ that is the same as $\mathbf{DB}$ except that every visited element is replaced with $\bot$. It is promised that *each real* addr *will be requested at most once.*

5.6.1 Definition

Let $\mathsf{params} = (m, \mathbf{w}, t_{\max})$ where m is the number of entries in the initial array $\mathbf{DB}$, $\mathbf{w}$ contains the bit-widths of the addr and val fields respectively, and $t_{\max}$ is the maximum number of times Read can be called.

- $\mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{InL}, \mathbf{OutL})$: receives the input and output labels $\mathbf{InL}$ and $\mathbf{OutL}$, and outputs the garbled circuitry GC;
- $\mathsf{Gmem} \leftarrow \mathsf{Init}^{\mathsf{GC}}(\{\!\{\mathbf{DB}\}\!\})$: receives a garbling of the initial memory $\{\!\{\mathbf{DB}\}\!\}$, and outputs the initial garbled memory Gmem;
- $\mathsf{Gmem}', \{\!\{\mathsf{val}\}\!\} \leftarrow \mathsf{Read}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{addr}\}\!\})$: receives a garbling of the logical address $\{\!\{\mathsf{addr}\}\!\}$ and if $\mathbf{DB}$ contains an element with the desired addr, return the garbled value of the element denoted $\{\!\{\mathsf{val}\}\!\}$; else if not found, simply return $\{\!\{\mathbf{0}\}\!\}$. In either case, the result should be garbled under using a τ-dependent output label specified by the list $\mathbf{OutL}$;
- $\{\!\{\mathbf{DB}'\}\!\} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{1\}\!\})$: upon receiving a garbled finalization signal $\{\!\{1\}\!\}$, returns $\{\!\{\mathbf{DB}'\}\!\}$. In particular, for any $i \in [0 : m)$, if $\mathbf{DB}[i]$ was not visited, then $\mathbf{DB}[i]' = \mathbf{DB}[i]$; else $\mathbf{DB}'[i] = (\bot, 0)$. Further, $\mathsf{Lbl}(\mathbf{DB}')$ should match the fixed finalization label contained in $\mathbf{OutL}$.

We often write $\mathbf{InL} := (\mathbf{InitL}, \mathbf{ReqL}, \mathbf{CtrlL})$ where the part $\mathbf{InitL}$ is consumed by Init, the part $\mathbf{ReqL}$ is consumed by Read, and the part $\mathbf{CtrlL}$ are the labels for the finalization signal. We often write $\mathbf{OutL} := (\mathbf{RdL}, \mathbf{FinL})$. where $\mathbf{RdL}$ denotes the labels used by the output of Read and $\mathbf{FinL}$ denotes the labels used by the output of $\mathsf{Finalize}$.

5.6.2 Construction

Our GBkt construction is depicted in Figure 7. The initialization function Init receives $\{\!\{\mathbf{DB}\}\!\}$ of length m; it first appends to the array $\{\!\{\mathbf{DB}\}\!\}$ a total of $t_{\max}$ garbled filler elements $\{\!\{\{(\bot_i, 0)\}_{i\in[0:t_{\max})}\}\!\}$

Evaluator	**Garbler**
• $\mathsf{Init}(\{\!\!\{\mathbf{DB}\}\!\!\})$:	
1. $\{\!\!\{\mathbf{DB}^*\}\!\!\} \leftarrow \mathsf{GPerm}(\{\!\!\{\mathbf{DB}\}\!\!\} \| \{\!\!\{\{(\bot_i, 0)\}_{i\in[0:t_{\max})}\}\!\!\})$;	create garbled circuit GPerm; create encodings $\{\!\!\{\{(\bot_i, 0)\}_{i\in[0:t_{\max})}\}\!\!\}$;
2. $\mathsf{GOTM.Init}(\{\!\!\{\mathbf{DB}^*\}\!\!\})$;	call $\mathsf{GOTM.Garble}$;
3. $\{\!\!\{\{\mathsf{addr}_i, i\}_{i\in[0:m+t_{\max})}\}\!\!\} \leftarrow \mathsf{GExtr}(\{\!\!\{\mathbf{DB}^*\}\!\!\})$; // $\mathsf{addr}_i = \mathbf{DB}^*[i].\mathsf{addr}$	create garbled circuit GExtr;
4. $\mathsf{GDict.Init}(\{\!\!\{\{\mathsf{addr}_i, i\}_{i\in[0:m+t_{\max})}\}\!\!\})$; // *$i$ is the "location" of* addr_i	call $\mathsf{GDict.Garble}$;
5. $\{\!\!\{\mathbf{Fidx}\}\!\!\} \leftarrow \mathsf{GFillers}(\{\!\!\{\mathbf{DB}^*\}\!\!\})$; // $\mathbf{Fidx}[j]$ = *the location of the filler* $\bot_j$	create garbled circuit $\mathsf{GFillers}$ which calls a sorting network to sort the fillers $\{\bot_i\}_{i\in[0:t_{\max})}$;
• $\mathsf{Read}(\{\!\!\{\mathsf{addr}\}\!\!\})$:	
1. $\{\!\!\{\mathsf{idx}'\}\!\!\} \leftarrow \mathsf{GDict.Lookup}(\{\!\!\{\mathsf{addr}\}\!\!\})$;	
2. $\mathsf{idx}, \{\!\!\{\mathsf{idx}\}\!\!\} \leftarrow \mathsf{GSel}(\{\!\!\{\mathsf{idx}'\}\!\!\}, \{\!\!\{\mathbf{Fidx}[\tau]\}\!\!\})$; // $\mathsf{idx} = \mathbf{Fidx}[\tau]$ *if* $\mathsf{idx}' = \bot$; *else* $\mathsf{idx} = \mathsf{idx}'$	for $\tau \in [0 : t_{\max})$, create garbled circuit GSel_τ.
3. $\{\!\!\{\_, \mathsf{val}\}\!\!\} \leftarrow \mathsf{GOTM.Read}(\{\!\!\{\mathsf{idx}^E\}\!\!\})$;	
4. output $\{\!\!\{\mathsf{val}\}\!\!\}$;	
• $\mathsf{Finalize}(\{\!\!\{1\}\!\!\})$: output $\mathsf{GOTM.Finalize}(\{\!\!\{1\}\!\!\})$.	

Figure 7: GBkt algorithm

prepared by the garbler. We assume that any filler element already in $\mathbf{DB}$ are encoded differently from these newly added fillers; further, we assume that each newly added filler has a unique identifier $\bot_i$ such that we can differentiate between them — to do this we can expand the bit width used to encode the addr field of each entry by $O(\log t_{\max})$ bits. It then applies a GPerm circuit to randomly permute this concatenated array, and obtains $\{\!\!\{\mathbf{DB}^*\}\!\!\}$. It then creates an access-revealing one-time memory GOTM initialized with $\{\!\!\{\mathbf{DB}^*\}\!\!\}$. Next, it creates a garbled dictionary GDict that stores the mapping between each addr and its physical location within $\{\!\!\{\mathbf{DB}^*\}\!\!\}$. Note that this GDict stores only metadata, and does not deal with actual data blocks. It also creates a list $\{\!\!\{\mathbf{Fidx}\}\!\!\}$ containing the locations of the $t_{\max}$ fillers $\{\bot_i\}_{i\in[0:t_{\max})}$. Later on, if the τ-th Read fails to find the element requested, the next filler $\bot_\tau$ whose location is stored in $\mathbf{Fidx}[\tau]$ will be read instead. It is guaranteed that each filler element will be read at most once.

During each Read request, the evaluator first looks up the GDict, and it uses GSel to post-process the garbled result returned by GDict. At this moment, if the addr requested exists in the GOTM, the evaluator obtains its position idx as well as a garbled version $\{\!\!\{\mathsf{idx}\}\!\!\}$; if not, then, idx will be the position of a random filler element. The evaluator now calls $\mathsf{GOTM.Read}(\{\!\!\{\mathsf{idx}^E\}\!\!\})$ to read the block stored at position idx of the one-time memory.

Henceforth, let $w_1 = |\mathsf{addr}|$ and let $w_2 = |\mathsf{val}|$. In our construction, $\mathsf{GDict}.|\mathsf{key}| = w_1$, $\mathsf{GDict}.|\mathsf{val}| = \log(t_{\max} + m)$, $\mathsf{GDict}.m = t_{\max} + m$, $\mathsf{GOTM}.m = t_{\max} + m$, $\mathsf{GOTM}.w = w_1 + w_2$. The total cost of our GBkt construction is $O(\widetilde{t} \cdot \lambda \cdot (w_1 \cdot \log^3 \widetilde{t} + w_2 \cdot \log^2 \widetilde{t} + \log^4 \widetilde{t})$ where $\widetilde{t} = t_{\max} + m$.

5.6.3 Security Proof

We will swap each garbled component one by one with a simulated counterpart. In our GBkt construction, all the garbled building blocks' input and output encodings are garblings (rather than sharings). Therefore, using our generalized notion of simulation (Definition 1), we do not have to strictly follow a reverse topological order for the swaps.

Consider the following sequence of hybrid experiments. For convenience, we write the proof *for the case when all the inputs are free inputs* — the case when there are some fixed inputs has a very similar proof.

- Hyb_0: the real-world view. In Hyb_0, the input labels of some subset of the garbled inputs $\{\!\!\{\mathbf{DB}\}\!\!\}$, $\{\{\!\!\{\mathsf{addr}\}\!\!\}_\tau\}_{\tau\in[0:t^*)}$, and $\{\!\!\{1\}\!\!\}_{t^*}$ are fixed. The output labels $\mathbf{OutL}$ are fixed. The labels of all other variables are chosen at random.
- Hyb_1: we first choose the labels for $\{\!\!\{\mathbf{DB}^*\}\!\!\}$ at random. We then choose the active values $\mathbf{DB}^* \leftarrow \mathcal{F}_{\mathrm{perm}}(\mathbf{DB}||\{(\bot_i, 0)\}_{i\in[0:t_{\max})})$ to be a random permutation of the array $\mathbf{DB}||\{(\bot_i, 0)\}_{i\in[0:t_{\max})}$. At this moment, the active encoding $\{\!\!\{\mathbf{DB}^*\}\!\!\}$ is determined. We may now replace $\mathsf{GPerm.GC}$ and the active encoding $\{\!\!\{\mathbf{DB}\}\!\!\}$ and $\{\!\!\{\{(\bot_i, 0)\}_{i\in[0:t_{\max})}\}\!\!\}$ with the output of $\mathsf{GPerm.Sim}$. The remaining garbled components and garbled free inputs are all generated using the honest algorithm. Note that knowing the cleartext inputs and the random permutation, the active values of all the remaining intermediate or final variables are uniquely determined, including the idx values output by GSel_τ in each time step is also uniquely determined.
- Hyb_2: otherwise identical to Hyb_1 except the following modification. Recall that each $\mathsf{GOTM.Read}$ call has two outputs addr and val. Choose all the active encodings for addr in all time steps at random. The active encodings of the val field in all time steps are already fixed since the $\mathbf{OutL}$ is fixed. Now, replace $\mathsf{GOTM.GC}$, $\mathsf{GOTM.Gmem}$, as well as its active garbled inputs $\{\{\!\!\{\mathsf{idx}\}\!\!\}_\tau\}_{\tau\in[0:t^*)}$, $\{\!\!\{1\}\!\!\}_{t^*}$ with the output of $\mathsf{GOTM.Sim}$ which takes in garbled (fixed subset of) inputs $\{\!\!\{\mathbf{DB}^*\}\!\!\}$ as well as the encodings of the active outputs $\{\{\!\!\{\mathsf{addr}, \mathsf{val}\}\!\!\}_\tau\}_{\tau\in[0:t^*)}$.
- Hyb_3: otherwise identical as Hyb_2 except the following modification. We now replace $\{\mathsf{GSel}_\tau.\mathsf{GC}$, $\{\!\!\{\mathsf{idx}'\}\!\!\}_\tau$, $\{\!\!\{\mathbf{Fidx}[\tau]\}\!\!\}_{\tau\in[0:t^*)}$ with the output of $\{\mathsf{GSel}_\tau.\mathsf{Sim}\}_{\tau\in[0:t^*)}$. For $\tau \in [t^*, t_{\max})$, replace $\mathsf{GSel}_\tau.\mathsf{GC}$ with simulated garbled circuits (see Remark 2).
- Hyb_4: otherwise identical as Hyb_3, except with the following modification. We now replace $\mathsf{GFillers.GC}$ with the output of $\mathsf{GFillers.Sim}$ which takes in the garbled output $\{\!\!\{\mathbf{Fidx}\}\!\!\}$ and the garbled input $\{\!\!\{\mathbf{DB}^*\}\!\!\}$.
- Hyb_5: otherwise identical as Hyb_4, except with the following modification. We now replace $\mathsf{GDict.GC}$, $\mathsf{GDict.Gmem}$, and the garbled inputs $\{\!\!\{\{\mathsf{addr}_i, i\}_{i\in[0:m+t_{\max})}\}\!\!\}$, $\{\{\!\!\{\mathsf{addr}\}\!\!\}_\tau\}_{\tau\in[0:t^*)}$ with the output of $\mathsf{GDict.Sim}$.
- Hyb_6: otherwise identical as Hyb_5, except with the following modification. We now replace $\mathsf{GExtr.GC}$ with the output of $\mathsf{GExtr.Sim}$ which takes in the garbled output $\{\!\!\{\{\mathsf{addr}_i, i\}_{i\in[0:m+t_{\max})}\}\!\!\}$ and the garbled input $\{\!\!\{\mathbf{DB}^*\}\!\!\}$.

Each time we make a swap in the above hybrid sequence, the computational indistinguishability of the adjacent pairs of hybrids rely on the security of the relevant garbled component that is being swapped. Finally, observe that in Hyb_6, every garbled component is being simulated. We can alternatively view Hyb_6 as follows:

1. We first choose the revealed idxes of each Read query to be random but non-overlapping positions.

2. We then choose the garbled outputs $\{\!\{\mathbf{DB}^*\}\!\}$ of GPerm at random, and simulate $\mathsf{GPerm.GC}$, $\{\!\{\mathbf{DB}\}\!\}$, and $\{\!\{\{(\bot_i, 0)\}_{i\in[0:t_{\max})}\}\!\}$ using $\mathsf{GPerm.Sim}$.
3. We then simulate the each of the remaining garbled components in the order of Hyb_3 to Hyb_6.

Using the above view, we no longer need to know the cleartext values of the inputs in the experiment; we only need to know the active garbled outputs and then we can invoke the simulators to complete the rest of the experiment.

5.7 Level Rebuilder (GRebuild)

Recall that in a non-recursive Bucket ORAM algorithm, $2\mathsf{B}$ denotes the bucket size, and w denotes the length of the payload string of each block. Suppose that n and B are powers of 2 and let $\ell_{\max} = \log_2 \frac{n}{\mathsf{B}}$. Recall that in a non-recursive Bucket ORAM of capacity n, we have $\ell_{\max}+1$ levels where each level $\ell \in [0 : \ell_{\max}]$ has 2^ℓ buckets, each of size $2\mathsf{B}$. Additionally, there is also a stash that can hold at most B blocks. Let t be the current time:

1. If $t+1$ is a multiple of n, then we need to merge the stash and levels $0, 1, \ldots, \ell_{\max}$ all into level $\ell_{\max}$. The levels $0, \ldots, \ell_{\max} - 1$ are emptied in this process. We henceforth use $\mathsf{bFinal} = 1$ to indicate this case.
2. Else, if $t + 1 = j \cdot \mathsf{B} \cdot 2^\ell$ for some odd integer j and some choice of ℓ, we need to merge the stash and levels $0, 1, \ldots, \ell - 1$ into level ℓ. Further, the levels $0, \ldots, \ell - 1$ are all emptied in the process. We use $\mathsf{bFinal} = 0$ to indicate this case.

A garbled rebuilder denoted $\mathsf{GRebuild}$ is simply a garbled circuit that performs this rebuilding. We may assume that each block is of the format $(\mathsf{addr}, \mathsf{leaf}, \mathsf{data})$. Let $\mathsf{params} = (\mathsf{B}, n, \mathbf{w})$ where $\mathbf{w}$ records the length of each of the fields addr, leaf, and data. Note that the parameter $\ell_{\max}$ is determined given B and n.

- $\mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \ell, \mathsf{bFinal}, \mathbf{InL}, \mathbf{OutL})$: takes in the relevant parameters params, bFinal, the destination level ℓ, the input and output labels denoted $\mathbf{InL}$ and $\mathbf{OutL}$ respectively, outputs the garbled circuitry GC.

- $\{\!\{\mathsf{level}'_i\}\!\}_{i\in[0:\ell]} \leftarrow \mathsf{Rebuild}^{\mathsf{GC}}(\{\!\{\mathsf{stash}\}\!\}, \{\!\{\mathsf{level}_i\}\!\}_{i\in[0:\ell)})$: used by the evaluator for rebuilding a level that is not the final level, i.e., if $\mathsf{bFinal} \neq 0$ during the Garble procedure. The algorithm takes in the garbled stash $\{\!\{\mathsf{stash}\}\!\}$, garbled levels $\{\!\{\mathsf{level}_i\}\!\}_{i\in[0:\ell)}$, and outputs new garbled levels $\{\!\{\mathsf{stash}'\}\!\}$ and $\{\!\{\mathsf{level}'_i\}\!\}_{i\in[0:\ell]}$ where level $i \in [0 : \ell]$ contains 2^i garbled buckets each of size $2\mathsf{B}$.

- $\{\!\{\mathsf{level}'_i\}\!\}_{i\in[0:\ell_{\max}]} \leftarrow \mathsf{Rebuild}^{\mathsf{GC}}(\{\!\{\mathsf{stash}\}\!\}, \{\!\{\mathsf{level}_i\}\!\}_{i\in[0:\ell_{\max}]})$: an alternative version of $\mathsf{Rebuild}$ that is used for the final level, i.e., if $\mathsf{bFinal} \neq 1$ during the Garble procedure. The algorithm takes in the garbled stash $\{\!\{\mathsf{stash}\}\!\}$, garbled levels $\{\!\{\mathsf{level}_i\}\!\}_{i\in[0:\ell_{\max}]}$, and outputs new garbled levels $\{\!\{\mathsf{stash}'\}\!\}$ and $\{\!\{\mathsf{level}'_i\}\!\}_{i\in[0:\ell_{\max}]}$ where level $i \in [0 : \ell_{\max}]$ contains 2^i garbled buckets each of size $2\mathsf{B}$.

The construction of $\mathsf{GRebuild}$ simply uses standard garbled circuits to garbled Bucket ORAM's rebuild algorithm. Suppose we use a linear-sized compaction circuit [AKL+20] to perform the $\mathsf{MergeSplit}$ operation that takes two buckets and outputs two buckets. Then, the size of this circuit is proportional to the total size of all the levels being rebuilt, i.e., $O(\mathsf{B} \cdot 2^\ell \cdot (|\mathsf{addr}| + |\mathsf{leaf}| + |\mathsf{data}|))$. Therefore, the size of the garbled circuit is $O(\lambda \cdot \mathsf{B} \cdot 2^\ell \cdot (|\mathsf{addr}| + |\mathsf{leaf}| + |\mathsf{data}|))$. In practice, we can use bitonic sort [Bat68] to implement the $\mathsf{MergeSplit}$ operation. In this case, the size of the garbled circuit is $O(\lambda \cdot \mathsf{B} \cdot 2^{\ell} \cdot (|\mathsf{addr}| + |\mathsf{leaf}| + |\mathsf{data}|) \cdot \log^2 \mathsf{B})$. For the purpose of getting the asymptotic performance of our final garbled RAM, it does not matter whether we use bitonic or compaction circuit to instantiate the **MergeSplit** operation, since this part of the overhead is not the dominating factor.

6 Non-Recursive Garbled Memory (NRGRAM)

6.1 Definition

A non-recursive garbled memory (NRGRAM) is almost an entire garbled memory, except that to access each logical addr, one has to provide a position identifier (both garbled and in cleartext) henceforth denoted $\{\!\{\mathsf{leaf}^E\}\!\}$, which specifies a path in the Bucket ORAM tree that the requested block resides on. More specifically, let $\mathsf{params} = (n, w, T)$ where n denotes the total number of blocks stored in the NRGRAM, w denotes the bit-width of each block's payload (not including metadata fields such as addr and leaf), and T denotes the maximum number of time steps. The call schedule is fixed a-priori: it must be a sequence of alternating requests ReadRm, Add, ReadRm, Add, ..., and in total there are T number of ReadRm operations and T number of Add operations.

A non-recursive garbled memory (NRGRAM) provides the following interface:

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{InL}^R, \mathbf{InL}^A, \mathbf{OutL})$: upon receiving the input labels $\mathbf{InL}^R$ for all the ReadRm calls and the input labels $\mathbf{InL}^A$ for all the Add calls, as well as the output labels **OutL** for the ReadRm calls, output GC and the initial Gmem;
- $\mathsf{Gmem}', \{\!\{\mathsf{rdata}\}\!\} \leftarrow \mathsf{ReadRm}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{addr}, \mathsf{leaf}^E\}\!\})$: upon receiving $\{\!\{\mathsf{addr}, \mathsf{leaf}^E\}\!\}$, output $\{\!\{\mathsf{rdata}\}\!\}$. If addr exists in the data structure and provided that $\{\!\{\mathsf{leaf}^E\}\!\}$ is a correct position identifier garbled under $\mathbf{InL}^R[t]$ where t denotes the local time, then rdata should be the value of the block at logical address addr; else if addr is not found, then $\mathsf{rdata} = \perp$. In either case, $\mathsf{Lbl}(\mathsf{rdata})$ should match $\mathbf{OutL}[t]$.
- $\mathsf{Gmem}' \leftarrow \mathsf{Add}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{addr}, \mathsf{leaf}, \mathsf{data}\}\!\})$: upon receiving a garbled block $\{\!\{\mathsf{addr}, \mathsf{leaf}, \mathsf{data}\}\!\}$, add it to the data structure. Henceforth, before addr is requested again, the block should reside on the path corresponding to leaf.

The local time t of a NRGRAM data structure is the number of times Add has been invoked (not counting the current invocation we are currently inside an Add call). Later in our full garbled RAM scheme, in every RAM step, each NRGRAM's ReadRM and Add functions will be each invoked once. Therefore, each NRGRAM's local time t coincides with the global time t of the garbled RAM, and each NRGRAM must support T calls which is the same as the RAM's maximum runtime. For this reason, we use the letter t to denote the NRGRAM's local time, and use T to denote the maximum number of time steps that must be supported.

Remark 4. We assume the first bit of the data field is used to encode whether the block is $\perp$. Specifically, if the first bit is 0, then the block is treated as $\perp$. We assume that when the honest evaluator calls $\mathsf{Add}(\{\!\{\mathsf{addr}, \mathsf{leaf}, \mathsf{data}\}\!\})$, the first bit of data is set to 1.

6.2 Data Structures and Labels

Without loss of generality, we may assume the capacity of the non-recursive ORAM tree n, the bucket capacity B, and the RAM's runtime T are all powers of 2. Let root be at level 0, and leaf be at level $\ell_{\max} := \log_2 \frac{n}{\mathsf{B}}$. We assume that the RAM program starts at time $t = 0$, and every time step the clock t increments by 1. Since in every RAM step, each non-recursive bucket ORAM is invoked once, the global time t also coincides with the non-recursive ORAM tree's local time step.

Garbled circuit inventory. All of the following garbled circuits are prepared by the garbler upfront in one shot. Each node at level ℓ in the tree has $T/(2^\ell \cdot \mathsf{B})$ instances (i.e., copies) of the following garbled circuitry: 1) $\mathsf{GSwitch}$ or $\mathsf{GLeafSwitch}$, and 2) GBkt. The instances are indexed from $0, 1, \ldots, T/(2^\ell \cdot \mathsf{B}) - 1$. During the fetch phase of time step $t \in [0 : T)$, the garbled instance indexed $\lfloor t/(2^\ell \cdot \mathsf{B}) \rfloor$ will be active.

During time step $t \in [0 : T - 2]$, if $(t + 1) \mod n = j \cdot (\mathsf{B} \cdot 2^\ell)$ where j is an odd integer, then there is some garbled circuitry that rebuilds levels $0, 1, \ldots, \ell$. In particular, if $\ell = \ell_{\max}$, then the rebuild takes as input garbled levels $0, 1, \ldots, \ell$ and outputs new garbled levels $0, 1, \ldots, \ell$; else, it takes garbled levels $0, 1, \ldots, \ell - 1$ and outputs new garbled levels $0, 1, \ldots, \ell$.

There are in total T/B instances of GStash, indexed by $0, 1, \ldots, T/\mathsf{B} - 1$. During time step t, the $\lfloor t/\mathsf{B} \rfloor$-th GStash instance is active.

Terminology. We shall use the notation GStash^t to denote the the GStash instance active at time t. We use the notation $\mathsf{GSwitch}^{V,t}$, $\mathsf{GLeafSwitch}^{V,t}$ or $\mathsf{GBkt}^{V,t}$ to denote the $\mathsf{GSwitch}$, $\mathsf{GLeafSwitch}$, or GBkt instance associated with tree node V and active at time t. Sometimes we represent a tree node $V = (i, j)$ which refers to the the j-th tree node in the i-th level. Using this notation, the same GStash, $\mathsf{GSwitch}$, or GBkt instance *may have multiple aliases*. Similarly, we use $\mathsf{GRebuild}^t$ to denote the $\mathsf{GRebulid}$ instance to be invoked at the end of time step t.

We say that $\mathsf{GSwitch}^{V,t}$ is the parent of $\mathsf{GSwitch}^{U,t}$ (or $\mathsf{GLeafSwitch}^{U,t}$) if V is a parent of U in the bucket ORAM tree; in this case, we also say that $\mathsf{GSwitch}^{U,t}$ or $\mathsf{GLeafSwitch}^{U,t}$ is a (left or right) child of $\mathsf{GSwitch}^{V,t}$. Note that these two $\mathsf{GSwitch}$ instances must be active at the same time for them to have a parent/child relationship. We often say that a switch instance $\mathsf{GSwitch}^{V,t}$ and a bucket instance $\mathsf{GBkt}^{V,t}$ are *paired* with each other — note that they are active at the same time t and belonging to the same tree node V.

Choosing labels. For each GStash, $\mathsf{GSwitch}$, $\mathsf{GLeafSwitch}$, and GBkt instance, the garbler chooses all of the labels (needed by the Garble procedures) at random, subject to the following constraints:

- $\mathsf{GSwitch}^{V,t}$ (or $\mathsf{GLeafSwitch}^{V,t}$) and its paired $\mathsf{GBkt}^{V,t}$ share the same address labels (for the $\{\!\{\mathsf{addr}\}\!\}$ inputs to the Read or Switch procedures) and finalization signal labels in all time steps. Moreover, $\mathsf{GBkt}^{V,t}.\mathbf{RdL} = \mathsf{GSwitch}^{V,t}.\mathbf{RdL}$ (or $\mathsf{GBkt}^{V,t}.\mathbf{RdL} = \mathsf{GLeafSwitch}^{V,t} = \mathbf{RdL}$ for the leaf level).
- The call at time t to $\mathsf{GSwitch}^{\mathrm{root},t}$ should adopt the input labels $\mathbf{InL}^R[t]$ (of the NRGRAM); further, GStash^t, $\mathsf{GBkt}^{\mathrm{root},t}$, and $\mathsf{GSwitch}^{\mathrm{root},t}$ share the same address labels (for the $\{\!\{\mathsf{addr}\}\!\}$ inputs to the Read or Switch procedure) in all time steps. Further, the call at time t to $\mathsf{GStash}^t.\mathsf{Add}$ should adopt the input labels $\mathbf{InL}^A[t]$ (of the NRGRAM);
- If non-leaf switches $\mathsf{GSwitch}_0$ and $\mathsf{GSwitch}_1$ are the left and right children of $\mathsf{GSwitch}$, then, for each $\tau \in [0 : 2\mathsf{B}]$, let

$$\mathsf{GSwitch}.\mathbf{OutL}_0[\tau] := \mathsf{GSwitch}_0.\mathbf{ReqL}[\tau] || \mathsf{GSwitch}_0.\mathbf{CtrlL}[\tau]$$
$$\mathsf{GSwitch}.\mathbf{OutL}_1[\tau] := \mathsf{GSwitch}_1.\mathbf{ReqL}[\tau] || \mathsf{GSwitch}_1.\mathbf{CtrlL}[\tau]$$

If leaf switches $\mathsf{GLeafSwitch}_0$ and $\mathsf{GLeafSwitch}_1$ are the left and right children of $\mathsf{GSwitch}$, and moreover, GBkt_0 and GBkt_1 are the two buckets associated with $\mathsf{GLeafSwitch}_0$ and $\mathsf{GLeafSwitch}_1$, respectively, then, for all $\tau \in [0:2\mathsf{B}]$, let[7]

$$\mathsf{GSwitch}.\mathbf{OutL}_0[\tau] := \mathsf{GBkt}_0.\mathbf{ReqL}[\tau]||\mathsf{GLeafSwitch}_0.\mathbf{InL}[\tau]||\mathsf{GBkt}_0.\mathbf{CtrlL}[\tau]$$
$$\mathsf{GSwitch}.\mathbf{OutL}_1[\tau] := \mathsf{GBkt}_1.\mathbf{ReqL}[\tau]||\mathsf{GLeafSwitch}_1.\mathbf{InL}[\tau]||\mathsf{GBkt}_1.\mathbf{CtrlL}[\tau]$$

- If $\mathsf{GSwitch}^{V,t}$ and $\mathsf{GSwitch}^{V,t+1}$ are not the same instance and they have the same children, then, let $\mathsf{GSwitch}^{V,t}.\mathbf{FinL} = \mathsf{GSwitch}^{V,t+1}.\mathbf{InitL}$ and let $\mathsf{GSwitch}^{V,t}.\mathbf{InitL} = \emptyset$ — in this case, our algorithm will not call $\mathsf{GSwitch}^{V,t}.\mathsf{Init}$ but will call $\mathsf{GSwitch}^{V,t+1}.\mathsf{Init}$.

For each level rebuilder instance denoted $\mathsf{GRebuild}^t$, let t be the time step at the end of which this rebuilder instance $\mathsf{GRebuild}^t$ is invoked — it must be that $(t+1) \mod n$ is an odd multiple of 2^ℓ. Suppose $\ell \neq \ell_{\max}$, i.e., the rebuild takes in levels $0, 1, \ldots, \ell-1$ and rebuilds levels $0, 1, \ldots, \ell$ — the case where $\ell = \ell_{\max}$ is similar. The garbler chooses the input and output labels of $\mathsf{GRebuild}^t$ as follows. For $i \in [0:\ell)$, let $\mathsf{GBkt}^{(i,0),t}, \ldots, \mathsf{GBkt}^{(i,2^i-1),t}$ be the garbled bucket instances active in level i at time t, and let GStash^t be the garbled stash active at time t; then, $\mathsf{GRebuild}.\mathbf{InL} = \mathsf{GStash}^t.\mathbf{FinL}||\{\mathsf{GBkt}^{(i,j),t}.\mathbf{FinL}\}_{i\in[0:\ell),j\in[0:2^i)}$. For $i \in [0:\ell]$, let $\mathsf{GBkt}^{(i,0),t+1}, \ldots, \mathsf{GBkt}^{(i,2^i-1),t+1}$ be the garbled bucket instances active in level i at time $t+1$, and let GStash^{t+1} be the garbled stash instance active at time $t+1$. Then, $\mathsf{GRebuild}^t.\mathbf{OutL} := \{\mathsf{GBkt}^{(i,j),t+1}.\mathbf{InitL}\}_{i\in[0:\ell],j\in[2^i]}$.

6.3 Construction

We describe our NRGRAM construction in Figure 8, where the relevant data structures and how to choose the encoding labels were explained earlier in Section 6.2. In the step marked ($\diamondsuit$), the same variables $\{\!\{\mathsf{leaf}\}\!\}$, $\{\!\{\mathsf{addr}\}\!\}$, $\llbracket L \rrbracket$ are overwritten by the outcome of the call to $\mathsf{GSwitch}^{V,t}.\mathsf{Switch}(\{\!\{\mathsf{leaf}\}\!\}, \{\!\{\mathsf{addr}\}\!\}, \llbracket L \rrbracket)$; keep in mind that the output variables do not have the same labels as the input variables, although the notation is the same.

The garbled data structures adopt the following parameters:

- For each GStash instance, the maximum number of operations $\mathsf{GStash}.m = \mathsf{B}$, and the word size $\mathsf{GStash}.w = w + \log_2 n$;
- For each $\mathsf{GSwitch}$ instance at level ℓ of the tree, let $\mathsf{GSwitch}.\mathsf{B} = \mathsf{B}$, and the addr field has bit width $\log_2 n$, the leaf field has bit width $\log_2 n - \ell$, the bit width of the L field has width $\lambda \cdot w$;
- Each $\mathsf{GLeafSwitch}$ instance is parametrized with the maximum number of invocations $\mathsf{GLeafSwitch}.t_{\max} = \mathsf{GLeafSwitch}.m = 2\mathsf{B}$ and the element bit-width $\mathsf{GLeafSwitch}.w = \lambda \cdot w$;
- Each GBkt instance adopts the parameters $\mathsf{GBkt}.m = \mathsf{GBkt}.t_{\max} = 2\mathsf{B}$, and the bit widths of the addr and val fields are $\log_2 n$ and w, respectively.

We now analyze the asymptotic performance of our NRGRAM scheme. One can easily verify that the dominating cost is incurred by the GBkt instances. The total cost of our NRGRAM is

$$\begin{aligned} &O(1) \cdot \frac{T}{\mathsf{B}} \cdot \log n \cdot \mathsf{B} \cdot \lambda \cdot \left(w \cdot \log^2 \mathsf{B} + \log n \cdot \log^3 \mathsf{B} + \log^4 \mathsf{B}\right) \\ =&O\left(T \cdot \log n \cdot \lambda \cdot (w + \log n \log \mathsf{B} + \log^2 \mathsf{B}) \cdot \log^2 \mathsf{B}\right) \end{aligned}$$

[7] The leaf switches take no $\{\!\{\mathsf{addr}\}\!\}$ nor $\mathsf{Finalize}$, and hence the parent $\mathsf{GSwitch}$ outputs $\{\!\{\mathsf{addr}\}\!\}$ or $\mathsf{Finalize}$ only to the children buckets (Figuer 8).

Evaluator

<u>$\mathsf{ReadRm}\left(\{\!\{\mathsf{addr}, \mathsf{leaf}^E\}\!\}\right)$:</u>

- if $t = 0$, then for every tree node V, call $\mathsf{GBkt}^{V,0}.\mathsf{Init}(\{\!\{\mathsf{bkt}_V^{\emptyset}\}\!\})$;
- $\_, \{\!\{\mathsf{rdata}_s\}\!\} \leftarrow \mathsf{GStash}^t.\mathsf{Read}(\{\!\{\mathsf{addr}\}\!\})$;
- $\llbracket L \rrbracket := \llbracket \mathbf{L}^*[t] \rrbracket$;
- For each node V in the tree from the root to leaf,
  - If V is not a leaf: let $\{\!\{\mathsf{leaf}\}\!\}, \{\!\{\mathsf{addr}\}\!\}, \llbracket L \rrbracket \leftarrow \mathsf{GSwitch}^{V,t}.\mathsf{Switch}(\{\!\{\mathsf{leaf}\}\!\}, \{\!\{\mathsf{addr}\}\!\}, \llbracket L \rrbracket)$; $(\diamondsuit)$
  - Else: $\mathsf{TrL} \leftarrow \mathsf{GLeafSwitch}^{V,t}.\mathsf{Switch}(\llbracket L \rrbracket)$;
  - $\{\!\{\mathsf{rdata}_\ell\}\!\} \leftarrow \mathsf{GBkt}^{V,t}.\mathsf{Read}(\{\!\{\mathsf{addr}\}\!\})$ where ℓ denotes the level of V;
- Let $\{\!\{\mathsf{rdata}\}\!\} := \mathsf{TrL} \oplus \{\!\{\mathsf{rdata}_s\}\!\} \oplus \left(\oplus_{\ell=0}^{\ell_{\max}} \{\!\{\mathsf{rdata}_\ell\}\!\}\right)$ and output $\{\!\{\mathsf{rdata}\}\!\}$.

<u>$\mathsf{Add}\left(\{\!\{\mathsf{addr}, \mathsf{leaf}, \mathsf{data}\}\!\}\right)$:</u>

- If $t+1 = T$, return; else continue with the following.
- Call $\mathsf{GStash}^t.\mathsf{Add}(\{\!\{\mathsf{addr}, \mathsf{leaf}, \mathsf{data}\}\!\})$;
- If $(t+1)$ is a multiple of n: invoke the garbled rebuilding algorithm similar to the case below marked $(\star)$, except that here, we shuffle levels $0, \ldots, \ell_{\max}$ into levels $0, \ldots, \ell_{\max}$;
- Else if $(t+1) \mod n = j \cdot (\mathsf{B} \cdot 2^\ell)$ for some odd integer j and some integer ℓ: $(\star)$
  - Let $\{\!\{\mathsf{stash}\}\!\} \leftarrow \mathsf{GStash}^t.\mathsf{Finalize}()$;
  - Call $\mathsf{RecFinalize}(\text{root}, \{\!\{1^*\}\!\}_t, \ell)$ described below;
  - For each level $i \in [0 : \ell)$, let $\{\!\{\mathsf{level}_i\}\!\} := \cup_{j \in [0:2^i)} \{\!\{\mathsf{bkt}_{i,j}\}\!\}$ where the variables $\{\!\{\mathsf{bkt}_{i,j}\}\!\}$ are output inside the $\mathsf{RecFinalize}$ call;
  - $\{\!\{\mathsf{level}_i\}\!\}_{i \in [0:\ell]} \leftarrow \mathsf{GRebuild}^t(\{\!\{\mathsf{stash}\}\!\}, \{\!\{\mathsf{level}_i\}\!\}_{i \in [0:\ell-1]})$;
  - For $i \in [0 : \ell]$, parse $\{\!\{\mathsf{level}_i\}\!\} := \{\!\{\mathsf{bkt}'_{i,j}\}\!\}_{j \in [0:2^i)}$; for $j \in [0 : 2^i)$, call $\mathsf{GBkt}^{(i,j),t+1}.\mathsf{Init}(\{\!\{\mathsf{bkt}'_{i,j}\}\!\})$;

<u>$\mathsf{RecFinalize}(V, \{\!\{1\}\!\}, \ell)$</u>

- $\{\!\{\mathsf{bkt}_V\}\!\} \leftarrow \mathsf{GBkt}^{V,t}.\mathsf{Finalize}(\{\!\{1\}\!\})$;
- If ℓ the leaf level, then return; else let $\{\!\{\mathsf{st}\}\!\}$ or $\{\!\{1_L, 1_R\}\!\} \leftarrow \mathsf{GSwitch}^{V,t}.\mathsf{Finalize}(\{\!\{1\}\!\})$; if $\mathsf{GSwitch}^{V,t+1}$ has the same children switches as $\mathsf{GSwitch}^{V,t}$, call $\mathsf{GSwitch}^{V,t+1}.\mathsf{Init}(\{\!\{\mathsf{st}\}\!\})$;
- Let U_L, U_R be the children of V, if the level of U_L and U_R is at most ℓ, call $\mathsf{RecFinalize}(U_L, \{\!\{1_L\}\!\}, \ell)$, $\mathsf{RecFinalize}(U_R, \{\!\{1_R\}\!\}, \ell)$.

Garbler

for every tree node V, let $\mathsf{bkt}_V^{\emptyset}$ be an array of 0s of appropriate length, create garbled state $\{\!\{\mathsf{bkt}_V^{\emptyset}\}\!\}$;

for $t \in [0 : T)$, let $\mathbf{L}^*[t] = \mathbf{OutL}[t] \oplus \mathsf{GStash}^t.\mathbf{OutL}[t \mod \mathsf{B}]$; create sharings $\llbracket \mathbf{L}^* \rrbracket := \{\mathbf{L}^*[t] \oplus K_t\}_{t \in [0:T)}$ where K_t should match the part of $\mathsf{GSwitch}^{\text{root},t}.\mathbf{InL}$ that is used for encoding the input $\llbracket L \rrbracket$ at time t;

call $\mathsf{GSwitch.Garble}$ for all $\mathsf{GSwitch}$ instances; call $\mathsf{GLeafSwitch.Garble}$ for all $\mathsf{GLeafSwitch}$ instances;

call $\mathsf{GBkt.Garble}$ for all GBkt instances;

call $\mathsf{GStash.Garble}$ for all GStash instances;

for every $t \in [0 : T)$ such that $t+1$ is a multiple of B, create a garbled finalization signal $\{\!\{1^*\}\!\}_t$ using the label $\mathsf{GSwitch}^{\text{root},t}.\mathbf{CtrlL}[\mathsf{B}]$;

call $\mathsf{GRebuild.Garble}$ for all $\mathsf{GRebuild}$ instances;

Figure 8: Non-Recursive Garbled RAM (NRGRAM) construction.

6.4 Proof of Correctness and Security

The following theorem states that our NRGRAM construction is secure for any *conforming* request sequence where "conforming" means that 1) every request generates a new random position identifier denoted leaf'_t; 2) every request comes with a correct position identifier leaf_t for the requested address addr_t, and moreover, if addr_t has not been requested before, then a random leaf_t is provided; 3) every Add request's addr' field is the same as the addr parameter passed to the previous ReadRm call.

Henceforth, we use $\overset{c}{\equiv}_\nu$ to denote the following: for any non-uniform p.p.t. adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$, such that the probability that $\mathcal{A}$ can distinguish the left-hand-side and the right-hand-side is upper-bounded by $\mathsf{negl}(\lambda) + \nu$. We will in fact use ν to encode the statistical failure probability in the following theorem statement.

Theorem 6.1 (Correctness and security of NRGRAM for conforming request sequences)**.** *Assume that all the garbled building blocks employed are secure. Let* $\nu = \exp(-\Omega(\mathsf{B})) \cdot \frac{T}{\mathsf{B}} \cdot \log \frac{n}{\mathsf{B}}$*. Then, there exists a p.p.t. simulator* Sim*, such that for for any* $\lambda \in \mathbb{N}$*, any* params*, any input sequence* $\{\mathsf{addr}_t, \mathsf{data}_t\}_{t\in[0:T)}$*, any output labels* $\mathbf{OutL}$ *of appropriate length,*

$$\begin{aligned} &\{\{\!\!\{\mathsf{rdata}_t\}\!\!\}\}_{t\in[0:T)},\ \mathsf{Gmem}, \mathsf{GC}, \{\!\!\{\mathsf{addr}_t, \mathsf{leaf}_t\}\!\!\}_{t\in[0:T)}, \{\!\!\{\mathsf{addr}'_t, \mathsf{leaf}'_t, \mathsf{data}_t\}\!\!\}_{t\in[0:T)} \\ \overset{c}{\equiv}_\nu\ &\mathcal{F}_{\mathrm{mem}}(\{\mathsf{addr}_t, \mathsf{data}_t\}_{t\in[0:T)}),\ \mathsf{Sim}\left(1^\lambda, \mathsf{params}, T, \{\!\!\{\mathsf{rdata}_t\}\!\!\}_{t\in[0:T)}, \{\mathsf{leaf}_t\}_{t\in[0:T)}\right) \end{aligned}$$

where

- $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)$, $\mathbf{InL}^R \overset{\$}{\leftarrow} \{0,1\}^{|\mathbf{InL}^R|}$, $\mathbf{InL}^A \overset{\$}{\leftarrow} \{0,1\}^{|\mathbf{InL}^A|}$;
- *for* $t \in [0:T)$*, let* $\mathsf{leaf}'_t \overset{\$}{\leftarrow} \{0,1\}^{|\mathsf{leaf}'|}$;
- *for* $t \in [0:T)$*, if* addr_t *does not exist in the data structure, let* $\mathsf{leaf}_t \overset{\$}{\leftarrow} \{0,1\}^{|\mathsf{leaf}|}$*; else let* $\mathsf{leaf}_t = \mathsf{leaf}'_r$ *where* $r < t$ *was the most recent time* addr_t *was added;*
- *let* $\{\!\!\{\mathsf{addr}_t, \mathsf{leaf}_t\}\!\!\}$ *be a correct garbling of* $(\mathsf{addr}_t, \mathsf{leaf}_t)$ *using the label* $\mathbf{InL}^R[t]$ *and let* $\{\!\!\{\mathsf{addr}'_t, \mathsf{leaf}'_t, \mathsf{data}_t\}\!\!\}$ *be a correct garbling of* $(\mathsf{addr}_t, \mathsf{leaf}'_t, \mathsf{data}_t)$ *using the label* $\mathbf{InL}^A[t]$*;*
- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{InL}^R, \mathbf{InL}^A, \mathbf{OutL})$*;*
- *for* $t \in [0:T)$*, let* $\{\!\!\{\mathsf{rdata}_t\}\!\!\} \leftarrow \mathsf{ReadRm}(\{\!\!\{\mathsf{addr}_t, \mathsf{leaf}_t\}\!\!\})$*, and call* $\mathsf{Add}(\{\!\!\{\mathsf{addr}'_t, \mathsf{leaf}'_t, \mathsf{data}_t\}\!\!\})$*; and*
- *finally,* $\mathcal{F}_{\mathrm{mem}}$ *is an ideal memory functionality that correctly outputs the result of each request.*

Further, the more generalized notion of simulation also holds, that is, there is a p.p.t. simulator Sim'*, such that if any subset of the garbled inputs* $\{\{\!\!\{\mathsf{addr}_t, \mathsf{leaf}_t\}\!\!\}, \{\!\!\{\mathsf{addr}'_t, \mathsf{leaf}'_t, \mathsf{data}_t\}\!\!\}\}_{t\in[0:T)}$ *are fixed and fed into a simulator* Sim'*, and* Sim' *otherwise receives the same inputs as* Sim*, then* Sim' *can simulate* $\mathsf{Gmem}, \mathsf{GC}$*, and the remaining garbled inputs that have not been fixed.*

Proof. Since the $\mathcal{F}_{\mathrm{mem}}$ functionality is deterministic, it suffices to prove correctness and security separately, i.e., we first prove that the real-world outputs match the ideal outputs except with $\nu/2$ probability (i.e., correctness), and then prove that the remaining terms on the left-hand and right-hand sides are computationally indistinguishable with the distinguishing probability upper bounded by $\nu/2 + \mathsf{negl}(\lambda)$. Note that for security, we need to prove two notions of simulation.

We first prove correctness. Let G be the good event that none of the buckets receive more than $2\mathsf{B}$ elements during rebuild. Let G' be the good event that every bucket receives at most $2\mathsf{B}$ number of Read requests during its life cycle. As long as both G and G' hold, correctness follows from the correctness of the underlying garbled building blocks. It suffices to show that G and G' hold except with $\nu/2$ probability. Due to a standard application of Chernoff bound, the probability that a fixed bucket receives more than $2\mathsf{B}$ elements during rebuild is upper bounded by $\exp(-\Omega_1(\mathsf{B}))$. Similarly, the probability that each fixed bucket receives more than $2\mathsf{B}$ requests during its life cycle is upper bounded by $\exp(-\Omega_1(\mathsf{B}))$ too. Taking union bound over all buckets, we get the desired statement.

We next prove security. Consider the following sequence of hybrid experiments. First, let Hyb_0 denote the real-world view.

Hyb_1**.** We first execute the underlying Bucket ORAM on the cleartext inputs $\{\mathsf{addr}_t, \mathsf{leaf}_t\}_{t\in[0:T)}$ and $\{\mathsf{addr}'_t, \mathsf{leaf}'_t, \mathsf{data}_t\}_{t\in[0:T)}$. If the aforementioned good events G and G' are not respected, we also abort outputting $\perp$. Else, we output the real-world view.

Claim 6.2. Hyb_1 *has statistical distance at most* $\nu/2$ *from* Hyb_0.

Proof. Follows directly from the above stochastic analysis showing that G and G' must hold except with $\nu/2$ probability. □

Hyb_2**.** Hyb_2 is otherwise identical to Hyb_1 except the following modification. For each $t \in [0:T)$, pick $\{\!\{\mathsf{rdata}_s\}\!\}_t$ $\{\!\{\mathsf{rdata}_0\}\!\}_t, \ldots, \{\!\{\mathsf{rdata}_{\ell_{\max}}\}\!\}_t$, and TrL_t at random subject to the constraint that $\{\!\{\mathsf{rdata}\}\!\}_t = \mathsf{TrL}_t \oplus \{\!\{\mathsf{rdata}_s\}\!\}_t \oplus_{\ell=0}^{\ell_{\max}} \{\!\{\mathsf{rdata}_\ell\}\!\}_t$. Hyb_2 is identically distributed as Hyb_1.

Next, we define a sequence of hybrid games denoted Hyb_*^t for every $t = k\mathsf{B}, (k-1)\mathsf{B}, \ldots, 0$ that is a multiple of B where $k = \frac{T}{\mathsf{B}} - 1$. Each hybrid Hyb_*^t will in turn consist of a sequence of inner hybrids where we make swaps one by one. Through this process, we will swap each garbled component with a simulated counterpart. The crux to making this argument work is to do the swaps in reverse topological order of the dependancy graph among these components. This way, whenever we are about to swap a garbled component with simulated, all of its outputs and a subset of its garbled inputs are already determined, and then we can swap the garbled circuitry, garbled memory, as well as the remaining garbled/shared inputs with simulated counterparts. To achieve this, roughly speaking, we need to make the swaps in reverse chronological order [Fin08] in terms of when each component is initialized or called.

Hyb_*^t**.** Let ℓ' be the least significant bit that is 0 in the binary representation of $t/\mathsf{B} - 1$, let $\ell = \min(\ell_{\max}, \ell')$. We make the following sequence of swaps:

- **Simulate the GBkt and $\mathsf{GSwitch}$ instances active at time t in levels $[0:\ell]$.** This needs to be done through a sequence of swaps in the reverse order of the nodes' levels in the tree. For $i = \ell$ down to 0, for every node (i,j) in level i of the tree,
  - replace $\mathsf{GBkt}^{(i,j),t}.\mathsf{GC}$, $\mathsf{GBkt}^{(i,j),t}.\mathsf{Gmem}$, and its garbled inputs with the output of $\mathsf{GBkt}^{(i,j),t}.\mathsf{Sim}$. Note that if $t = k\mathsf{B}$, we can simulate the garbled finalization outputs of all these bucket instances at random (before calling the GBkt instance's simulator); otherwise the garbled finalization outputs should already be determined by a future $\mathsf{GRebuild}$ simulator.
  - replace $\mathsf{GSwitch}^{(i,j),t}.\mathsf{GC}$, $\mathsf{GSwitch}^{(i,j),t}.\mathsf{Gmem}$, and its remaining garbled/shared inputs with the output of $\mathsf{GSwitch}^{(i,j),t}.\mathsf{Sim}$ — note that if i is at the leaf level, the corresponding $\mathsf{GLeafSwitch}$ instance should be used instead of $\mathsf{GSwitch}$.

  To make these swaps, we need to know at which global time steps t the relevant GBkt and $\mathsf{GSwitch}$ instances are invoked. This knowledge can be easily inferred from the leakage which contains the leaf identifiers of every ReadRm request.

- **Simulate the GStash^t instance.** Replace $\mathsf{GStash}^t.\mathsf{GC}$, $\mathsf{GStash}^t.\mathsf{Gmem}$, and its remaining garbled inputs (including all garbled inputs $\{\!\{\mathsf{addr}, \mathsf{leaf}, \mathsf{data}\}\!\}$ to the Add call with the output of $\mathsf{GStash}^t.\mathsf{Sim}$.
- **Simulate the $\mathsf{GRebuild}^{t-1}$ instance.** Replace $\mathsf{GRebuild}^{t-1}.\mathsf{GC}$ and its garbled inputs with the output of $\mathsf{GRebuild}^{t-1}.\mathsf{Sim}$.

One can verify that whenever we swap a garble component to simulated, its garbled/shared outputs have already been determined; further, if the garbled component shares garbled inputs with other garbled components that are already simulated then part of its garbled inputs may be determined too. Some parts of garbled inputs may also be determined since they correspond to the part of garbled inputs of the NRGRAM that are fixed. Thus, we can call the corresponding garbled component's simulator to simulate the garbled circuitry, garbled memory, and the remaining garbled/shared inputs. Each time we make a swap, we can argue that the views before and after the swap are computationally indistinguishable due to the security of the relevant garbled component. □

7 Final Construction: Garbled Memory (GRAM)

7.1 Definition

Let $\mathsf{params} = (N, W, T)$ where N denotes the total number of blocks stored in GRAM, W denotes the bit-width of each block, and T denotes the maximum number of time steps. A garbled RAM (GRAM) scheme provides the following syntax:

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{InL}, \mathbf{OutL})$: upon receiving the input labels $\mathbf{InL}$ and the output labels $\mathbf{OutL}$, output GC and the initial Gmem;
- $\mathsf{Gmem}', \{\!\{\mathsf{rdata}\}\!\} \leftarrow \mathsf{Access}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{\mathsf{addr}\}\!\}, \{\!\{\mathsf{wdata}\}\!\}, \{\!\{\mathsf{bWrite}\}\!\})$: the input consists of the garbled logical address requested $\{\!\{\mathsf{addr}\}\!\}$, a garbled bit $\{\!\{\mathsf{bWrite}\}\!\}$ indicating whether the operation is a write, and and the new data to overwrite with $\{\!\{\mathsf{wdata}\}\!\}$. If addr has been written before, the result rdata should be equal to the last value written; else the result $\mathsf{rdata} = 0$. Further, $\mathsf{Lbl}(\{\!\{\mathsf{rdata}\}\!\}) = \mathbf{OutL}[t]$ where t denotes the current time step.

Remark 5. Just like in Section 6, we shall assume that the first bit of the payload field is used to encode whether the block is $\perp$. Specifically, if the first bit is 0, then block is $\perp$. We assume that the honest evaluator calls $\mathsf{Access}(\{\!\{\mathsf{addr}, \mathsf{wdata}, \mathsf{bWrite}\}\!\})$, the first bit of wdata is always set to 1.

7.2 Construction

We use a standard recursion technique first suggested in Oblivious RAM constructions [SCSL11, SSS12]. Let $D = \log_2 N$, our construction creates $D+1$ instances of NRGRAM denoted NRGRAM_0, …, NRGRAM_D, of geometrically growing sizes. For each $d \in [0 : D]$, the d-th instance stores $\mathsf{NRGRAM}_d.n = 2^d$ blocks. Except for NRGRAM_D which stores real data blocks, all other instances store metadata. More specifically, for $d \in [0 : D]$, we have $\mathsf{NRGRAM}_d.w = 2 \log N$, and $\mathsf{NRGRAM}_D.w = W$ where W is the size of an actual data block. For $d \in [1 : D-1]$, blocks in NRGRAM_d have logical addresses that are d bits long. Suppose that we want to know the position identifier in the final instance NRGRAM_D of some data block with the logical address addr. To do this, we need to look up the metadata block with the address $\mathsf{addr}[0 : D-2]$ in NRGRAM_{D-1}, and inside this metadata block we store a pair of position identifiers (p_0, p_1), where p_b is the position

Evaluator

```
$\underline{\mathsf{Access}(\{\!\{\mathsf{addr}\}\!\}, \{\!\{\mathsf{wdata}\}\!\}, \{\!\{\mathsf{bWrite}\}\!\}):}$
• $\{\!\{\mathsf{leaf}^1\}\!\}, \mathsf{leaf}^1 \leftarrow \mathsf{GCSel}^0(\{\!\{p_0^0, p_1^0\}\!\}, \{\!\{\mathsf{addr}[0]\}\!\}, \{\!\{\mathbf{rL}_*^0[t]\}\!\})$;
  // $\beta = \mathsf{addr}[0]$; if $p_\beta^0 = \bot$, $\mathsf{leaf}^1 = \mathbf{rL}_*^0[t]$, else $\mathsf{leaf}^1 = p_\beta^0$
• for $d \in [1 : D)$                                  // let $D = \log_2 N$
  1. $\{\!\{p_0^d, p_1^d\}\!\} \leftarrow \mathsf{NRGRAM}^d.\mathsf{ReadRm}(\{\!\{\mathsf{addr}[0 : d)\}\!\}, \{\!\{\mathsf{leaf}^d\}\!\})$;
  2. $\{\!\{\mathsf{leaf}^{d+1}\}\!\}$, $\mathsf{leaf}^{d+1} \leftarrow \mathsf{GCSel}^d(\{\!\{p_0^d, p_1^d\}\!\}, \{\!\{\mathsf{addr}[d]\}\!\}, \{\!\{\mathbf{rL}_*^d[t]\}\!\})$;
     /* $\beta = \mathsf{addr}[d]$; if $p_\beta^0 = \bot$, then $\mathsf{leaf}^{d+1} = \mathbf{rL}_*^d[t]$, else $\mathsf{leaf}^{d+1} = p_\beta^d$ */
• $\{\!\{\mathsf{rdata}\}\!\} \leftarrow \mathsf{NRGRAM}^D.\mathsf{ReadRm}(\{\!\{\mathsf{addr}\}\!\}, \{\!\{\mathsf{leaf}^D\}\!\})$;
• $\{\!\{\mathsf{data}\}\!\} \leftarrow \mathsf{GCSel}^*(\{\!\{\mathsf{rdata}\}\!\}, \{\!\{\mathsf{wdata}\}\!\}, \{\!\{\mathsf{bWrite}\}\!\})$;
  // $\mathsf{data} = \mathsf{rdata} \cdot (1 - \mathsf{bWrite}) + \mathsf{wdata} \cdot \mathsf{bWrite}$;
• $\mathsf{NRGRAM}^D.\mathsf{Add}(\{\!\{\mathsf{addr}\}\!\}, \{\!\{\mathbf{rL}^D[t]\}\!\}, \{\!\{\mathsf{data}\}\!\})$
• for $d = D-1, \ldots, 1$:
  1. $\{\!\{\widetilde{p}_0^d, \widetilde{p}_1^d\}\!\} \leftarrow \mathsf{GCUpd}^d(\{\!\{p_0^d, p_1^d\}\!\}, \{\!\{\mathsf{addr}[d]\}\!\}, \{\!\{\mathbf{rL}^{d+1}[t]\}\!\})$;
     // $\widetilde{p}_b^d = \mathbf{rL}^{d+1}[t] \cdot (\mathsf{addr}[d] \oplus b \oplus 1) + p_b^d \cdot (\mathsf{addr}[d] \oplus b)$;
  2. $\mathsf{NRGRAM}^d.\mathsf{Add}(\{\!\{\mathsf{addr}[0 : d)\}\!\}, \{\!\{\mathbf{rL}^d[t]\}\!\}, \{\!\{\widetilde{p}_0^d, \widetilde{p}_1^d\}\!\})$;
• $\{\!\{p_0^0, p_1^0\}\!\} \leftarrow \mathsf{GCUpd}^0(\{\!\{p_0^0, p_1^0\}\!\}, \{\!\{\mathsf{addr}[0]\}\!\}, \{\!\{\mathbf{rL}^1[t]\}\!\})$;
  // $p_b^0 = \mathbf{rL}^1[t] \cdot (\mathsf{addr}[0] \oplus b \oplus 1) + p_b^0 \cdot (\mathsf{addr}[0] \oplus b)$;
• output $\{\!\{\mathsf{data}\}\!\}$;
```

Garbler

```
create initial garbled states $\{\!\{p_0^0, p_0^1\}\!\} = \{\!\{\bot, \bot\}\!\}$; for $t \in [0 : T)$, sample a random $\mathbf{rL}_*^0[t]$ and create the garbled state $\{\!\{\mathbf{rL}_*^0[t]\}\!\}$;
for $t \in [0 : T)$, create garbled circuit $\mathsf{GCSel}_t^0$;
for $d \in [1 : D]$, call $\mathsf{NRGRAM.Garble}$;
for $t \in [0 : T)$, for $d \in [1 : D)$, sample a random $\mathbf{rL}_*^d[t]$ and create the garbled state $\{\!\{\mathbf{rL}_*^0[t]\}\!\}$; create garbled circuit $\mathsf{GCSel}_t^d$;
for $t \in [0 : T)$, create garbled circuit $\mathsf{GCSel}_t^*$;
for $t \in [0 : T)$, for $d \in [1 : D]$, sample a random $\mathbf{rL}^d[t]$ and create the garbled state $\{\!\{\mathbf{rL}^d[t]\}\!\}$;
for $t \in [0 : T)$, for $d \in [0 : D)$, create the garbled circuit $\mathsf{GCUpd}_t^d$;
```

Figure 9: Full Garbled RAM algorithm

identifier of the block at address $\mathsf{addr}[0 : D-2]||b$ in the final instance NRGRAM_D. To read the block at address $\mathsf{addr}[0 : D-2]$ in NRGRAM_{D-1}, we need to recursively look up its own position identifier. To do this, we need to look up in NRGRAM_{D-2} a metadata block of address $\mathsf{addr}[0 : D-3]$, and inside this metadata block we also store a pair (p'_0, p'_1) where p'_b is the position identifier of the block at address $\mathsf{addr}[0 : D-3]||b$ in NRGRAM_{D-2}, and so on. After we successfully read the data block at address addr in NRGRAM_D, we relocate the block to a new path in the underlying ORAM tree. We need to inform the parent instance NRGRAM_{D-1} of this new position identifier, which in turn triggers a sequence of updates to all NRGRAM instances. The full construction is described in Figure 9. Specifically, the garbler will choose upfront all the random position identifiers $\mathbf{rL}^d[t]$ to be consumed in each time step t by the each instance NRGRAM_d, garble these random choices, and include them in the garbled memory. Similarly, the garbler also prepares garbled states $\mathbf{rL}_*^d[t]$ which are to be consumed if the address looked up is not found in NRGRAM_d where $d \in [0 : D)$, in which case the next instance NRGRAM_{d+1} will look up a random path identified by leaf^{d+1} in the underlying ORAM tree.

The total cost of our GRAM construction is

$$\begin{aligned} & O\left(T \cdot \log N \cdot \lambda \cdot \log^2 \mathsf{B} \cdot (W + \log N(\log N \cdot \log \mathsf{B} + \log^2 \mathsf{B} + \log N))\right) \\ = & O\left(T \cdot \log N \cdot \lambda \cdot \log^2 \mathsf{B} \cdot \left(W + \log^2 N \log \mathsf{B} + \log N \log^2 \mathsf{B}\right)\right) \end{aligned}$$

If we let δ denote the statistical failure probability, then, the above expression can be simplified to the following where $\widetilde{O}(\cdot)$ hides $\mathsf{poly}\log\log \frac{T \cdot N}{\delta}$ factors:

$$\widetilde{O}\left(T \cdot \lambda \cdot (W \log N + \log^3 N)\right)$$

In other words, the amortized cost of each memory access is $\widetilde{O}\left(\lambda \cdot (W \log N + \log^3 N)\right)$.

7.3 Proof of Correctness and Security

Theorem 7.1 (Correctness of our GRAM scheme)**.** *Our* GRAM *scheme (Figure 9) achieves correctness with probability* $1 - \delta$ *where* $\delta = \exp(-\Omega(\mathsf{B})) \cdot \frac{T \log N}{\mathsf{B}} \cdot \log \frac{N}{\mathsf{B}}$.

Proof. Observe that in our GRAM algorithm, the sequence of calls to every underlying NRGRAM respects the constraints of Theorem 6.1. Under such conforming inputs, recall that each underlying NRGRAM has $\nu = \exp(-\Omega(\mathsf{B})) \cdot \frac{T}{\mathsf{B}} \cdot \log \frac{n}{\mathsf{B}}$ error probability. Taking a union bound over all $O(\log N)$ NRGRAM instances, our GRAM construction's correctness failure probability is upper bounded by $\delta = \exp(-\Omega(\mathsf{B})) \cdot \frac{T \log N}{\mathsf{B}} \cdot \log \frac{N}{\mathsf{B}}$. □

Theorem 7.2 (Security of our GRAM scheme)**.** *Suppose that* N, T, W *are polynomially bounded in the security parameter* λ*, and moreover, we adopt* $\mathsf{B} = \omega(\log \lambda)$ *to instantiate the underlying* NRGRAM *instances. Then, the resulting* GRAM *scheme (Figure 9) respects Definition 1.*

Proof. We prove for the case where all inputs are free inputs — the case when there are some fixed inputs enjoys a very similar proof. Our proof goes through a sequence of hybrid experiments where we swap each garbled component one by one with a simulated counterpart.

- Real: the real-world view.
- Hyb_*: otherwise identical as Real except that we now use the following randomized process for sampling the labels for the input and intermediate variables. Choose $\{\mathbf{rL}^d_*\}_{d \in [0:D)}$ and $\{\mathbf{rL}^d\}_{d \in [1:D]}$ at random. At this moment, if we know all the cleartext addresses $\{\mathsf{addr}_t\}_{t \in [0:T)}$ requested, we can uniquely determine each leaf^d_t for $d \in [0 : D]$ and for $t \in [0 : T)$. Choose $\{\{\!\{\mathsf{rdata}\}\!\}_t\}_{t \in [0:T)}$ at random (which fixes the labels of these variables). Choose the active encodings $\{\{\!\{\mathsf{leaf}^D\}\!\}_t\}_{t \in [0:T)}$ at random. All the remaining undetermined labels for inputs and intermediate variables are chosen at random. Hyb_* is identically distributed as Hyb_0.
- Hyb_D: consists of the following inner hybrids in which we swap garbled components one by one with simulated counterparts:
  - Replace $\mathsf{NRGRAM}_D.\mathsf{GC}$, $\mathsf{NRGRAM}_D.\mathsf{Gmem}$, and the inputs $\{\{\!\{\mathsf{addr}, \mathbf{rL}^D[t], \mathsf{data}\}\!\}_t\}_{t \in [0:T)}$ with the output of $\mathsf{NRGRAM}_D.\mathsf{Sim}$, which takes in $\{\{\!\{\mathsf{rdata}\}\!\}_t\}_{t \in [0:T)}$, the garbled fixed set of inputs $\{\{\!\{\mathsf{leaf}^D\}\!\}_t\}_{t \in [0:T)}$, and the leakage $\{\mathsf{leaf}^D_t\}_{t \in [0:T)}$.
  - Replace $\{\mathsf{GCSel}^*_t.\mathsf{GC}\}_{t \in [0:T)}$, $\{\{\!\{\mathsf{wdata}, \mathsf{bWrite}\}\!\}_t\}_{t \in [0:T)}$ with the outcome of $\{\mathsf{GCSel}^*_t.\mathsf{Sim}\}_{t \in [0:T)}$ which takes in garbled outputs $\{\{\!\{\mathsf{data}\}\!\}_t\}_{t \in [0:T)}$ and the garbled inputs $\{\{\!\{\mathsf{rdata}\}\!\}_t\}_{t \in [0:T)}$.

- $\mathsf{Hyb}_{D-1}, \ldots, \mathsf{Hyb}_0$: for each $d = D - 1$ down to 0, Hyb_d in turn contains a sequence of inner hybrids where each hybrid is otherwise identical to the previous one, except with the following modifications:
  - For each $t \in [0 : T)$, replace $\mathsf{GCSel}^d_t.\mathsf{GC}$, and its inputs $\{\!\{p^d_0, p^d_1\}\!\}_t$, $\{\!\{\mathbf{rL}^d_*[t]\}\!\}$ with the output of $\mathsf{GCSel}^d_t.\mathsf{Sim}$, which takes in $\{\!\{\mathsf{leaf}^{d+1}\}\!\}, \mathsf{leaf}^{d+1}$ and garbled input $\{\!\{\mathsf{addr}[d]\}\!\}$ as input;
  - Replace $\mathsf{NRGRAM}_d.\mathsf{GC}$, $\mathsf{NRGRAM}_d.\mathsf{Gmem}$, and the inputs $\{\{\!\{\mathsf{addr}, \mathbf{rL}^d[t], \mathsf{data}\}\!\}_t\}_{t\in[0:T)}$ with the output of $\mathsf{NRGRAM}_d.\mathsf{Sim}$, which takes in $\{\{\!\{p^d_0, p^d_1\}\!\}_t\}_{t\in[0:T)}$, the garbled fixed set of inputs $\{\{\!\{\mathsf{leaf}^d, \mathsf{addr}[0 : d)\}\!\}_t\}_{t\in[0:T)}$, and the leakage $\{\mathsf{leaf}^d_t\}_{t\in[0:T)}$.
  - For each $t \in [0 : T)$, replace $\mathsf{GCUpd}^d_t.\mathsf{GC}$ with the output of $\mathsf{GCUpd}^d_t.\mathsf{Sim}$, which takes in all of its garbled inputs and outputs as input;

  From Hyb_D to Hyb_0, with each swap, the computational indistinguishability of the adjacent hybrids rely on the security of the garbled component being swapped.

- Ideal. In the Ideal experiment, we sample all the leakages leaf^d_t for $t \in [0 : T)$ and $d \in [0 : D]$ at random. We then sample all the encodings $\{\!\{\mathsf{leaf}^d_t\}\!\}$ at random too. We then call the simulator for each garbled component and simulate them one by one, in the order of Hyb_D to Hyb_0.

Clearly, Ideal is identically distributed as Hyb_0. Moreover, in Ideal, we no longer need to know the cleartext inputs to GRAM; we only need to know the active encodings of the outputs. □

8 Practical Optimizations and Concrete Performance

8.1 Practical Optimizations

We adopt the following practical optimizations and parameters when evaluating the concrete performance of our scheme.

Optimization 1: use permutation rather than sorting during rebuild. Our first optimization is inspired by an elegant idea described by Heath et al. [HKO21]. Except the last level $\ell_{\max}$ in the NRGRAM, we can actually replace the garbled sort with garbled permutation GPerm during rebuild, and we instantiate the garbled permutation with Waksman's construction [Wak68]. Observe that the garbler generates all the random coins used in the rebuild. Henceforth, imagine that the real blocks read are not immediately marked as filler until the last level $\ell_{\max}$ is rebuilt — note that our stochastic bounds hold for this case too. Now, we refer to a block by the time it was added, until the next time $\ell_{\max}$ is rebuilt, the garbler knows a-priori where each block is in the hierarchy of levels. Only for the last level $\ell_{\max}$ do we use oblivious sorting and we instantiate the sort with Bitonic sort [Bat68] which gives good concrete performance for small problem sizes. Note that the last level is special since it is rebuilt "in-place", and we need to compress by a factor of 2 when rebuilding it.

Optimization 2: bucket capping and merged overflow pile. In our experiment, we want to guarantee a *satistical* security parameter[8] of at least 60 per memory access. To this end, we

[8] The *statistical* security parameter should be treated differently from the *computational* security parameter λ. The statistical security parameter captures graceful degradation of privacy, whereas if the computational security parameter is too small, the scheme can be completely broken. Typically, a statistical security parameter between $[40, 80]$ is considered reasonable. In our evaluation, we set the computational security parameter $\lambda = 128$ as is typical in the standard literature.

adopt two sets of parameters, either bucket size 256, and the expected load of the bucket 128; or bucket size 156 and the expected load of the bucket 64 — our simulator automatically selects the better one of these two configurations depending on the block size. Since our buckets are small, in practice, we do not instantiate them with the asymptotical construction described in Section 5.6. For small problem sizes, the naïve approach of linear scan has better concrete performance. Using linear-scan garbled buckets, our simulation shows that the garbled buckets account for the majority of the cost at kilobyte to gigabyte data scales.

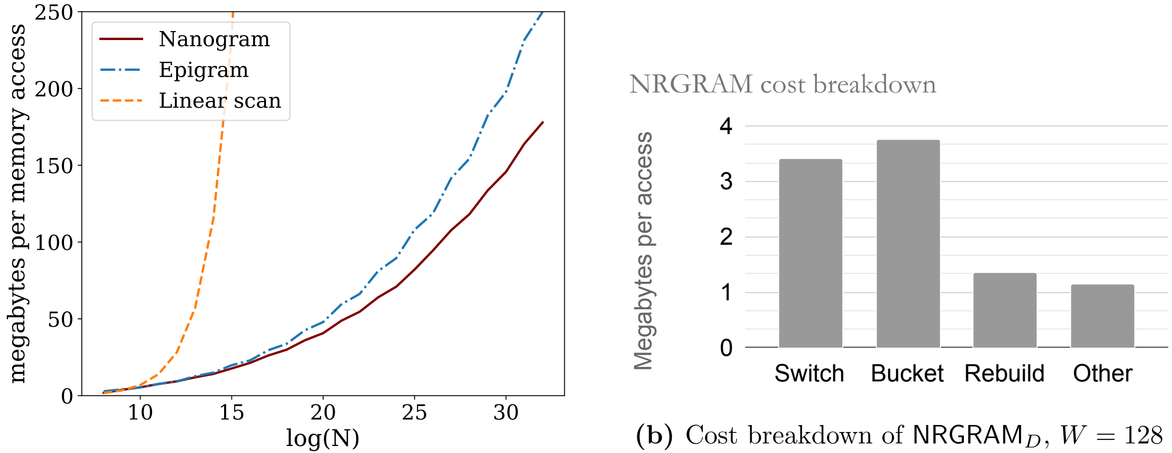
We devise a new idea that cuts the cost of the garbled buckets by a constant factor. Just like in the first optimization, we will again rely on the fact that the garbler actually knows the data movement schedule in the hierarchy of levels ahead of time (except for the last level). Therefore, we can cap each bucket at some a-priori fixed capacity S that is smaller than the bucket size. In this case, there is a small probability that each bucket may overflow, but the overflow probability is not small enough to meet our desired statistical security parameter. We can put all overflowing blocks into a global overflow pile. Since the number of overflowing elements per bucket has a sharp geometric tail bound, the global overflow pile is not too large with very high probability. We use simulation to find the right size of overflow pile to meet our desired statistical security parameter. In particular, in our simulation, we plot the size of the overflow pile against $\log_2(\frac{1}{\delta})$ where δ is the probability that the overflow pile exceeds the specified size. Since the tail bound is exponentially sharp, the plot shows up as a straightline, and we extrapolate it to the point where $\log_2(\frac{1}{\delta}) \approx 65$, such that even after taking a union bound over all other failure probabilities, the final statistical security parameter per memory access is 60 or higher. This optimization is also compatible with optimization 1, since the garbler knows the data movement schedule ahead of time except for the last level. With this optimization, the rebuild circuits also need to take the garbled overflow pile as input.

Additional optimizations. Through our experiments, we find that a recursion fanout of 4 achieves the best concrete performance, i.e., each metadata block in the recursion structure stores the position identifiers of 4 entries in the next recursion depth. We stop the recursion and use a linear-scan garbled RAM instead whenever the problem size becomes small enough such that linear-scan becomes more efficient. We applied this optimization to both our NanoGRAM and EpiGRAM [HKO21].

For some of the key components that are called repeatedly such as garbled stack, we made hand-crafted optimizations to the circuit structure to optimize the performance by a constant, in comparison with the descriptions in earlier works [ZE13, HKO21]. Again, we applied the same optimization to both NanoGRAM and EpiGRAM.

8.2 Concrete Performance

A simulator of NanoGRAM. To evaluate the concrete performance, we made a simulator of NanoGRAM. Our simulator can count the following quantities for each garbled component 1) the number of AND gates needed for the garbled circuitry GC; 2) the number and sizes of operations of the type $\{\!\{b^E\}\!\} \cdot [\![y]\!] \to [\![b^E \cdot y]\!]$; 3) the number of garblings and sharings needed for creating the initial garbled memory Gmem; and 4) any other information needed, e.g., encryptions and translation labels. Using the state-of-the-art garbled circuit scheme [RR21] to construct the garbled circuits, and using the efficient implementation of $\{\!\{b^E\}\!\} \cdot [\![y]\!] \to [\![b^E \cdot y]\!]$ type gates, we can calculate the total communication needed. Specifically, as Rosulek and Roy showed in their elegant work [RR21], there is a garbling scheme for boolean circuits in which XOR gates are free and each AND gate requires $1.5\lambda + 5$ bits of communication. Further, the elegant work of Heath



(a) Comparison with linear scan GRAM and EpiGRAM: block size $W = 128$ bits.

(b) Cost breakdown of NRGRAM_D, $W = 128$ bits, $N = 2^{15}$.

Figure 10: Concrete performance of NanoGRAM.

et al. [HKO21] showed that we can garble a $\{\!\{b^E\}\!\} \cdot [\![y]\!] \rightarrow [\![b^E \cdot y]\!]$ type gate using only $|y|$ bits assuming that $|y|$ is a multiple of λ.

A simulator to estimate a lower bound of EpiGRAM. We made another simulator to give a lower bound of the concrete performance of EpiGRAM [HKO21]. Unlike our NanoGRAM simulator which is high-fidelity and accounts for every gate needed in an actual implementation, our simulator of EpiGRAM is intended to give a lower bound of EpiGRAM's performance — *we are willing to give EpiGRAM an unfair advantage in our comparison.* In particular, our EpiGRAM simulator accounted only for the concrete costs that is the dominating terms in their asymptotical bound $O(W \log^2 N + \log^4 N)$. We ignored various circuit components whose costs are asymptotically smaller than $O(W \log^2 N + \log^4 N)$. Therefore, we expect that the actual performance of EpiGRAM in a full-fledged implementation will be slightly worse than the curves shown in our charts.

Our EpiGRAM simulator implemented all the concrete optimizations documented in the Heath et al. work [HKO21]. Beyond these optimizations, it is outside the scope of our work to further optimize their scheme. However, for every optimization that we made to NanoGRAM that also applies to EpiGRAM, we did apply the same optimization to EpiGRAM. Therefore, overall, *our simulation actually made more optimizations to their scheme than those documented in their paper.*

Concrete performance. In Figure 10a, we compared the performance of NanoGRAM with that of the naïve linear-scan GRAM as well as EpiGRAM [HKO21], where the word size $W = 128$ bits. In NanoGRAM, since the parameter B (i.e., average load per bucket) has to be at least 64 or 128 to get a reasonable statistical security parameter, the smallest N we used in our experiment is 2^8. Just like EpiGRAM, we start to outperform the naïve linear-scan GRAM at about $N = 2^9$. Our concrete performance is on par with EpiGRAM at small choices of N, but at about $N = 2^{13}$, we start to outperform EpiGRAM, and as shown in the figure, the improvement is of an asymptotical nature — the larger the N, the greater our speedup.

Figure 10b shows the cost breakdown for the NRGRAM for the final data level. The breakdown suggests that the garbled buckets are the most costly, whereas the garbled switches closely follow. This plot also shows the motivation for our optimization 2 — had we not performed this optimization, the total garbled bucket cost would be more than 2× higher than the total garbled switch cost.

Acknowledgments

This work is in part supported by a DARPA SIEVE grant, a Packard Fellowship, NSF awards under the grant numbers 2128519 and 2044679, and a grant from ONR. We gratefully acknowledge Wenting Zheng for helpful technical discussions during an early phase of the project.

References

[AKL$^+$20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. In *Eurocrypt*, 2020.

[App13] Benny Applebaum. Garbling xor gates "for free" in the standard model. In *TCC*, 2013.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings*, pages 307–314, 1968.

[CCC$^+$16] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In *ITCS*, pages 179–190. ACM, 2016.

[CCHR16] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled RAM or: How to delegate your database. In *TCC (B2)*, volume 9986 of *Lecture Notes in Computer Science*, pages 61–90, 2016.

[CH16] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In *ITCS*, pages 169–178. ACM, 2016.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-xor" technique. In *TCC*, 2012.

[CNS18] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In *TCC (2018)*, volume 11240 of *Lecture Notes in Computer Science*, pages 636–668. Springer, 2018.

[CS17] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In *TCC (2)*, volume 10678 of *Lecture Notes in Computer Science*, pages 72–107. Springer, 2017.

[CSLN21] T.-H. Hubert Chan, Elaine Shi, Wei-Kai Lin, and Kartik Nayak. Perfectly oblivious (parallel) RAM revisited, and improved constructions. In *ITC*, volume 199 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.

[Fin08] David Fincher. The curious case of benjamin button, film, 2008.

[FNR$^+$15] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth Oblivious RAM. Cryptology ePrint Archive, Report 2015/1065, 2015. https://ia.cr/2015/1065.

[GGH$^+$13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PETS*, 2013.

[GHL$^+$14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In *EUROCRYPT*, pages 405–422, 2014.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In *FOCS*, pages 210–229. IEEE Computer Society, 2015.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled ram from one-way functions. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 449–458, New York, NY, USA, 2015. Association for Computing Machinery.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[Gol87] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[HK20] David Heath and Vladimir Kolesnikov. Stacked garbling - garbled circuit proportional to longest execution path. In *CRYPTO (2)*, volume 12171 of *Lecture Notes in Computer Science*, pages 763–792. Springer, 2020.

[HK21a] David Heath and Vladimir Kolesnikov. Logstack: Stacked garbling with O(b log b) computation. In *EUROCRYPT (3)*, volume 12698 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2021.

[HK21b] David Heath and Vladimir Kolesnikov. One hot garbling. In *CCS*, pages 574–593. ACM, 2021.

[HKO21] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. Practical garbled RAM: GRAM with $O(\log^2 n)$ overhead. Cryptology ePrint Archive, Report 2021/1519, 2021. https://ia.cr/2021/1519.

[HL20] Carmit Hazay and Mor Lilintal. Gradual gram and secure computation for ram programs. In *Security and Cryptography for Networks*, pages 233–252. Springer, 2020.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for XOR gates that beats free-xor. In *CRYPTO (2)*, volume 8617 of *Lecture Notes in Computer Science*, pages 440–457. Springer, 2014.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP*, 2008.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *CRYPTO*, 2018.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble ram programs. In *EUROCRYPT*, 2013.

[LO17] Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In *CRYPTO (2)*, volume 10402 of *Lecture Notes in Computer Science*, pages 66–92. Springer, 2017.

[PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious ram with logarithmic overhead. In *FOCS*, 2018.

[RR21] Mike Rosulek and Lawrence Roy. Three halves make a whole? beating the half-gates lower bound for garbled circuits. In *CRYPTO (1)*, volume 12825 of *Lecture Notes in Computer Science*, pages 94–124. Springer, 2021.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, jan 1968.

[WCS15] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*, 2015.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *CCS*, 2014.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[ZE13] Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *IEEE S & P*, 2013.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250. Springer, 2015.

A Additional Building Blocks

A.1 Slow Queue (GSlowQueue)

We describe a garbled linear-scan queue. This building block has been used in past works [ZRE15, HKO21], so we supply the details here only for completeness. Let $\mathsf{params} = (m, w, t_{\max})$ where m denotes the number of elements in the initial database $\mathbf{DB}$ and also the maximum number of elements, w denotes the bit-width of each element, and $t_{\max}$ is the maximum number of operations. A slow queue supports two operations Pop and Push, each controlled by a flag $b \in \{0, 1\}$. If $b = 1$, then a real Pop or Push operation happens; otherwise, nothing happens. Initially, the queue is

Evaluator

```
• Pop($\{\!\{b\}\!\}$):
1. $\llbracket \mathsf{res} \rrbracket \leftarrow \mathsf{GMul}_\tau(\llbracket \mathbf{D}[0] \rrbracket, \{\!\{b\}\!\})$;   // $\mathsf{res} = \mathbf{D}[0] \cdot b$
2. $\{\!\{\mathbf{bTail}\}\!\} \leftarrow \mathsf{GLShift}_\tau(\{\!\{\mathbf{bTail}\}\!\}, \{\!\{b\}\!\})$;
3. $\llbracket \mathbf{D} \rrbracket \leftarrow \mathsf{GLShift'}_\tau(\llbracket \mathbf{D} \rrbracket, \{\!\{b\}\!\})$;
   // if $b = 1$, left shift by 1
4. return $\llbracket \mathsf{res} \rrbracket$.
• Push($\{\!\{b\}\!\}, \llbracket x \rrbracket, \llbracket y \rrbracket$):
  For $\llbracket z \rrbracket \in \{\llbracket x \rrbracket, \llbracket y \rrbracket\}$, do:
1. $\{\!\{\mathbf{bTail}\}\!\} \leftarrow \mathsf{GRShift}_\tau(\{\!\{\mathbf{bTail}\}\!\}, \{\!\{b\}\!\})$;
   // if $b = 1$, right shift by 1
2. $\llbracket z' \rrbracket \leftarrow \mathsf{GMul}_\tau(\llbracket z \rrbracket, \{\!\{b\}\!\})$;
3. $\llbracket \mathbf{S} \rrbracket \leftarrow \mathsf{GMul'}_\tau(\llbracket z' \rrbracket, \{\!\{\mathbf{bTail}\}\!\})$ // $\mathbf{S} = z' \cdot \mathbf{bTail}$
4. $\llbracket \mathbf{D} \rrbracket \leftarrow \mathsf{GAdd}_\tau(\llbracket \mathbf{D} \rrbracket, \llbracket \mathbf{S} \rrbracket)$        // $\mathbf{D} = \mathbf{D} \oplus \mathbf{S}$
```

Garbler

Create a sharing $\llbracket \mathbf{D} \rrbracket$ of $\mathbf{DB}||0^w$, create a garbling $\{\!\{\mathbf{bTail}\}\!\}$ of the initial bitvector $\mathbf{bTail} = 0^{m-1}||1||0$;
For any $\tau \in [0 : t_{\max} - 1]$ that is a Pop step, create the garbled circuit $\mathsf{GLShift}_\tau$, $\mathsf{GLShift}'_\tau$, and GMul_τ;

For $\tau \in [0 : t_{\max} - 1]$ that is a Push step, create two copies of each of the following garbled circuits: $\mathsf{GRShift}_\tau$, GMul_τ, GMul'_τ, and GAdd_τ;

Figure 11: GSlowQueue algorithm.

assumed to be empty. We make the following assumptions: 1) the schedule of operations is known at garbling time, i.e., the garbler knows in which time step which operation will be invoked; 2) it is guaranteed that at runtime, Pop or and Push will not cause the stack to underflow or overflow.

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{DB}, \mathbf{InL}, \mathbf{OutL})$: takes in an initial database $\mathbf{DB}$, input and output labels $\mathbf{InL}$ and $\mathbf{OutL}$, and outputs $\mathsf{Gmem}, \mathsf{GC}$.
- $\mathsf{Gmem}', \llbracket \mathsf{res} \rrbracket \leftarrow \mathsf{Pop}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{b^E\}\!\})$: if $b = 1$, then pop the element at the head of the queue denoted res; else, $\mathsf{res} = 0$. Furthermore, the $\mathsf{Lbl}(\llbracket \mathsf{res} \rrbracket)$ must match the output label of time τ stored in $\mathbf{OutL}$.
- $\mathsf{Gmem}' \leftarrow \mathsf{Push}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{b^E\}\!\}, \llbracket x \rrbracket, \llbracket y \rrbracket)$: if $b = 1$, push the elements x and y into the (logical) queue; else do not push anything.

Construction. GSlowQueue can be constructed by naïvely performing $O(1)$ garbled linear scans of the queue upon every request, as shown in Figure 11. In this algorithm, we can use the efficient garbled circuit proposed by Heath et al. [HKO21] for computing $\{\!\{b^E\}\!\} \cdot \llbracket y \rrbracket \rightarrow \llbracket b \cdot y \rrbracket$ where $b^E \in \{0,1\}$ is known to the evaluator, and $y \in \{0,1\}^k$ is hidden. It is not hard to see that their algorithm can be extended to the case when the garbler is told what input labels $\{\!\{b^E\}\!\}$ and $\llbracket y \rrbracket$ should use, and what output label $\llbracket b \cdot y \rrbracket$ should use.

A.2 A One-Use Stack (GOTStack)

For constructing our access-revealing one-time memory (Section 5.5), we need a special stack that receives a single Push operation upfront, and receives at most one Pop operation afterwards before Finalize is called.

Evaluator	**Garbler**
• $\mathsf{Push}(\{\!\{x\}\!\})$: store $\{\!\{x\}\!\}$;	
• $\mathsf{Pop}(\{\!\{b\}\!\})$:	
1. $\{\!\{y\}\!\} \leftarrow \mathsf{GSel}(\{\!\{x\}\!\}, \{\!\{b\}\!\})$; // $y = (1-b)\cdot x + b \cdot \bot$ 2. $\mathsf{GVault.Skip}()$;	Create GSel garbled circuit; Call $\mathsf{GVault.Garble}$ with $\{\!\{0\}\!\}$ which is garbled under the label I_{pop}.
• $\mathsf{Finalize}(\{\!\{1\}\!\})$:	
1. if $\tau = 0$: $\{\!\{0\}\!\} \leftarrow \mathsf{GVault.Finalize}(\{\!\{1\}\!\})$; $\mathsf{Pop}(\{\!\{0\}\!\})$; 2. Output $\{\!\{y\}\!\}$.	

Figure 12: $\mathsf{GOTStack}$ algorithm.

- $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, w, (I_{\text{push}}, I_{\text{pop}}, O_{\text{pop}}, F, C_0, C_1))$: takes in input labels I_{push} for the single Push call, input and output labels I_{pop} and O_{pop} for the single Pop call, output label F for the $\mathsf{Finalize}$ call, and signal labels C_0 and C_1, where C_0 is used if the stack is finalized without any Pop calls; and C_1 is used if the stack is finalized after a Pop call.

- $\mathsf{Gmem}' \leftarrow \mathsf{Push}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{x\}\!\})$: takes in a garbled element $\{\!\{x\}\!\}$ garbled under the label I_{push} and push it into the logical stack;

- $\mathsf{Gmem}', \{\!\{\mathsf{res}\}\!\} \leftarrow \mathsf{Pop}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{b^E\}\!\})$: depending on the flag b, either pop an element from the stack or do nothing. Correctness requires that if $b = 1$, then $\mathsf{res} = x$, else $\mathsf{res} = 0$. Moreover, it must be that $\mathsf{Lbl}(\llbracket \mathsf{res} \rrbracket) = O_{\text{pop}}$;

- $\{\!\{\mathsf{res}\}\!\} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{1\}\!\})$: upon receiving a finalization signal $\{\!\{1\}\!\}$ garbled either under C_0 or C_1 depending on whether Pop has been called before, output a garbling of res, where $\mathsf{res} = x$ if the element has not been popped before; else $\mathsf{res} = \bot$. Further, $\mathsf{Lbl}(\{\!\{\mathsf{res}\}\!\}) = F$.

Construction. We give a construction of $\mathsf{GOTStack}$ in Figure 12.

B Blackbox Construction from One-Way Functions

Our full construction of garbled memory, GRAM assumes a circular correlation-robust hash function [ZRE15, Definition 1], which is a typical assumption for efficient garble circuits. In this section, we construct a garbled memory that employs *only one-way functions* in a blackbox way. Intuitively, this construction is a modification of our former construction. Specifically, we now mechanically instantiate each building block using Yao's garbling [Yao86, Yao82] (instead of Free-XOR garbling), and moreover we replace all sharings with garblings. This new instantiation introduces an extra λ factor in cost, but is still more efficient compared to previous blackbox one-way function (OWF) based constructions [GLO15, LO17]. Specifically, our blackbox OWF-based construction incurs $\widetilde{O}\left(\lambda^2 \cdot (W \log N + \log^3 N)\right)$ cost per memory access whereas the cost of previous constructions can be found in Table 1 of Section 1. We now sketch our blackbox OWF-based construction.

Terminologies of Yao's garbled circuits. Recall that in Yao's garbling scheme, each wire is associated with a pair of two secret keys sampled uniformly at random, and the two keys represent values of 0 and 1 correspondingly. Each gate is associated with ciphertexts that encrypt each possible output of the gate represented as a secret key, and thus the gate can be evaluated when the appropriate input keys are provided. For each wire, we say the pair of two keys (k_0, k_1) is the *language* of the wire, and when a binary value b (either 0 or 1) is given on the wire, we call corresponding key for b the *garbling of* b, denoted as $\{\!\{b\}\!\}$. When the value b is fixed, we also call it a 0-garbling or 1-garbling. For multiple bits on a vector of wires, we also extend the terminology and say the language or the garbling of the wires or bits.

In our construction, we will use a *re-garbling* circuit. The re-garbling circuit takes in a garbled bit $\{\!\{b\}\!\}$ and a garbled language $\{\!\{L = (k_0, k_1)\}\!\}$, and then it outputs the garbling k_b of value b. Later without ambiguity, the garbling k_b will also denote by $\{\!\{b\}\!\}$ in the context, but it is encoding in another language.

Differences from our earlier construction. The construction mostly follows our former construction, but there are a couple major difference. First, we replace all sharings with Yao's garblings. In particular, recall that in our earlier construction, we forward a *sharing* of a global-time-dependent language along a tree path in our GOTM construction. In our OWF-based construction, we will instead forward a *garbling* of the global-time-dependent language along the tree path. Along each step of the way, the garbling is translated to the local-time-dependent language of each tree node receiving it. This is the reason why the extra λ is taken in the cost: garbling of a language increases the string by λ times. See the garbled switch for details.

The second difference is in the non-recursive garbled memory. In our OWF-based construction, we forward a garbling of the XOR of all blocks read from all buckets so far along the tree path. In particular, since every garbled bucket now outputs a garbling of the block fetched, we can no longer directly use the XOR trick to XOR all of them together to obtain the final fetch result for the entire path. One strawman solution is for each garbled bucket to output its fetched result garbled under a global-time-dependent label; however, we would then have to forward the garblings of logarithmically many global-time languages along the tree path, which would an additional logarithmic factor. Our approach of forwarding along a garbling of the accumulated result avoids this drawback. See Appendix B.2 for details.

B.1 Building Blocks

We refer the reader to Section 4 and 5 for the inputs and syntax of our building blocks, and we focus on the modifications and the new costs of the constructions below. The only exception is that the garbled switch is functionally different from that of Section 5.3. We also use the same notation as in Section 4 and 5: the computational security parameter is denoted by λ, the size of each element is denoted by w, the number of elements is denoted by n or m, and B is the statistical security parameter (which will be chosen later in the garbled RAM).

EpiGRAM [HKO21] based on one-way function. The construction of Heath et al. [HKO21] can be modified to use only one-way functions as follows. In their lazy permutation network, the internal nodes in the tree can be realized similar to our $\mathsf{GSwitch}$ later in this section, which costs $O(n \log^2 n \cdot \lambda \cdot (\lambda w + \log n))$, where the λw term comes from storing Yao's languages in the stack. Their leaf nodes cost $O(n\lambda \cdot (\lambda w \log n + \log^2 n))$ because each node stores $\log n$ pairs of "pointer and language" (that takes $(\lambda w + \log n)$ bits). Then, their ORAM and maintenance cost is $O(n \log^2 n \cdot \lambda(w + \log n))$. Composing these together yields a "non-recursive" garbled RAM, which costs $O(n\lambda \log^2 n \cdot (\lambda w + \log n))$ per n accesses (on n memory words, each of w bits). The full construction needs $\log n$ levels of recursion, each recursion stores data in words of $\log n$ bits except for the first recursion level. Hence, the overall cost is

$$O(n\lambda \log^2 n \cdot (\lambda w + \log n)) + \log n \cdot O(n\lambda \log^2 n(\lambda \log n)) = O(n\lambda^2 \log^2 n \cdot (w + \log^2 n))$$

per n accesses, and that is $O(\lambda^2 \log^2 n \cdot (w + \log^2 n))$ as claimed earlier in this section.

Garbled Stash (GStash) Our construction of GStash is the same as Section 4.1. We create a length-m garbled array which is initialized with all $\{\!\{0\}\!\}$s. To read an element, we use a linear scan circuit to scan through the entire array. To add an element, we write the element into the desired index. This construction has the same cost as Section 4.1, namely $O(\lambda \cdot w \cdot m)$ bits.

Slow Blackbox Garbled Dictionary (GDict) Our GDict construction is basically identical to the GDict described in Section 4.2, except we replace the encoded *sharings* with garblings. Recall that the construction follows in two steps: first, we build a binary search tree with the keys, where each key maps to a value in a garbled array. On a lookup, the evaluator routes his request through the search tree, finding the value with the associated key. Second, we hide the access patterns of the search tree by using standard oblivious data structure and recursion techniques.

Recall from Section 4.2 that the routing can be accomplished using two pop-only garbled stacks. In our construction, these stacks can now be instantiated using Yao's garblings (instead of sharings), which means that each w-bit language is garbled into a garbling of size $\lambda \cdot w$ bits. Everything else in the construction remains identical, hence this construction costs $O(\lambda^2(w \log^2 m + \log^4 m))$ bits per memory access to implement a garbled memory of size m and block size w.

Using this garbled memory and the recursion and oblivious data structure techniques discussed in Section 4.2, our GDict construction thus costs GDict cost of $O(t_{\max} \cdot \lambda^2 \cdot (w_1 \log^3 m + w_2 \log^2 m + \log^4 m))$ where $w_1 = |\mathsf{key}|$, $w_2 = |\mathsf{val}|$, and $t_{\max}$ is the maximum number of queries.

Garbled Random Permutation(GPerm) Our GPerm construction is the same as Section 4.3. We use Waksman's permutation network, which has a cost of $O(\lambda \cdot w \cdot m \cdot \log m)$.

Expiring Vault (GVault)**.** The construction is almost identical to that described in Section 5.1, where the only difference is now we instantiate Yao's garbling circuits (hence, the input languages $\mathbf{InL} = (C_0, \ldots, C_{t_{\max}})$ shall consist of a pair of 0-garbling and 1-garbling for each C_i, but only those 1-garblings are used). The cost is the same, $O(t_{\max} \cdot (\lambda + w))$.

Slow Queue ($\mathsf{GSlowQueue}$)**.** Recall that $\mathsf{params} = (m, w, t_{\max})$ denotes the maximum number of elements, the bit-width of each element, and the maximum number of operations. The construction is similar to that of Appendix A.1, but the elements are pushed, popped, or stored using Yao's garbling (instead of sharings). That is, each w-bit element is garbled into an encoding of $\lambda \cdot w$ bits, and the element takes $O(\lambda \cdot w)$ during the push, store, then pop procedures. The cost is increased by λ, which is $O(\lambda m w)$ per push or pop operation.

Stack (GStack)**.** Next, we consider the pop-only stack constructed in Section 5.2. Similarly, $\mathsf{params} = (m, w, t_{\max})$ denotes the maximum number of elements, the bit-width of each element, and the maximum number of operations. The construction is identical to Figure 3 except that all $(\mathsf{GSlowQueue}_\ell)_{\ell \in [0:\ell_{\max}]}$ are instantiated using Yao's garbling as described in this section. The construction takes overall cost $O(\lambda w t_{\max} \cdot \log m)$ bits.

Switch ($\mathsf{GSwitch}$)**.** The switch from Yao's garbling is conceptually similar to that of Section 5.3, but now the output is re-garbled using the languages that are popped from one of the two stacks, where the re-garbling makes the main difference. Specifically, the procedure to re-garble the output of Switch follows standard Yao's garbling for both the garbler and the evaluator. That is, the input is a garbled bit $\{\!\{b\}\!\}$ and a garbling of the language $\{\!\{K\}\!\}$, where the wire of b is represented with a pair of garblings (L^0, L^1) such that $\{\!\{b\}\!\} = L^b$, the language K consists of a pair of secret keys (K^0, K^1) representing 0 or 1 respectively; taking as input $\{\!\{b\}\!\}$ and $\{\!\{K\}\!\}$, we want to reveal K^b as the re-garbling of b. To do so, the garbler prepares a garbled circuit that computes $b \cdot K^0 \oplus (1-b) \cdot K^1$ and output the result in the clear, where the circuit size is linear in the input size. We extend this to a string of bits and K is a sequence of languages, and we say it is *re-garbling* $\{\!\{b\}\!\}$ *under language* K when $\{\!\{K\}\!\}$ is given.

Another major difference is the message forwarded by the switch. Recall that the switch of Section 5.3 is designed to forward the encoding of $L' = L \oplus \mathbf{RdL}[\tau]$ (to the child node), where L is encoded in the input but $\mathbf{RdL}[\tau]$ is hardwired in the switch (Figure 4). However in this section, the switch now forwards only $L' = L$. Looking forward, it is because in the construction of non-recursive garbled RAM, we will aggregate and pass the read data from each bucket to the leaf *through* the switches (Appendix B.2).

For completeness, we present the modified procedure in Figure 13. The construction in Figure 13 costs $O(\mathsf{B}\lambda \cdot (w_1 + w_2) \cdot \log \mathsf{B})$ bits, where $w_1 = |\mathsf{leaf}| + |\mathsf{addr}|$ and $w_2 = |L|$.

Garbled One-Time Memory (GOTM) Our construction of GOTM is identical to the one presented in Section 5.5, but now composed of the GStack and $\mathsf{GSwitch}$ data structures presented above. For the sake of completeness, the GOTM algorithm using Yao's garbling is presented in Figure 14.

Our GOTM construction has incurs a cost of λ over the construction presented in Section 5.5, due to the blowup incurred from garbling rather than sharing the languages. Our construction thus costs $O(m \cdot \lambda^2 \cdot (w + \log m) \cdot \log^2 m)$ bits.

Bucket (GBkt)**.** The construction is almost identical to that of Section 5.6, where the only difference is that the dictionary (GDict) and the one-time memory (GOTM) are instantiated using the variants of this section. Recall that a bucket stores m pairs of $(\mathsf{addr}, \mathsf{val})$ where $|\mathsf{addr}| = w_1$ and $|\mathsf{val}| = w_2$. Following the construction and the analysis, the total cost of our GBkt construction is the summation of GDict and GOTM, that is,

$$\begin{aligned} O\left(\widetilde{t} \cdot \lambda^2 \cdot (w_1 \log^3 \widetilde{t} + \log^4 \widetilde{t})\right) &+ O\left(\widetilde{t} \cdot \lambda^2 \cdot (w_2 + \log \widetilde{t}) \cdot \log^2 \widetilde{t}\right) \\ &= O\left(\widetilde{t} \cdot \lambda^2 \cdot (w_1 \log^3 \widetilde{t} + w_2 \log^2 \widetilde{t} + \log^4 \widetilde{t})\right), \end{aligned}$$

where $\widetilde{t} = t_{\max} + m$.

Level Rebuilder ($\mathsf{GRebuild}$)**.** Recall that a level rebuilder takes as input a stash and ℓ levels, where each level $i < \ell$ consists of 2^i buckets, each bucket or the stash consists of B number of elements, and each element is a tuple $(\mathsf{addr}, \mathsf{leaf}, \mathsf{data})$; the output is a new level ℓ (consisting of 2^ℓ buckets) and new empty levels $i < \ell$ (see Section 5.7 for details). Using Yao's garbling, the construction is composed identically as in Section 5.7, and the cost is $O(\lambda\mathsf{B} \cdot 2^\ell \cdot (|\mathsf{addr}| + |\mathsf{leaf}| + |\mathsf{data}|))$.

Evaluator	**Garbler**
• $\mathsf{Init}(\{\!\{\mathsf{st}^E_\emptyset\}\!\})$: *// called when* $\mathbf{InitL} \neq \emptyset$	
1. parse $\{\!\{\mathsf{st}^E_\emptyset\}\!\} := \{\!\{\beta_{b,i}\}\!\}_{b\in\{0,1\}, i\in[0:2\mathsf{B})}$; 2. for $b \in \{0,1\}$, $i \in [0 : 2\mathsf{B})$, call $\mathsf{GStack}_b.\mathsf{Pop}(\{\!\{\beta_{b,i}\}\!\})$;	Create two garbled stacks GStack_0 and GStack_1 as explained in a separate subroutine;
• $\mathsf{Switch}(\{\!\{\mathsf{leaf}^E\}\!\}, \{\!\{\mathsf{addr}\}\!\}, \{\!\{L\}\!\})$:	
1. Call $\{\!\{\beta_0 = \mathsf{leaf}[0]\}\!\}, \{\!\{\beta_1 = 1-\beta_0\}\!\}, \{\!\{\mathsf{leaf}' = \mathsf{leaf}[1:]\}\!\}, \{\!\{\mathsf{addr}' = \mathsf{addr}\}\!\}, \{\!\{L' = L\}\!\} \leftarrow \mathsf{GCSw}_\tau(\{\!\{\mathsf{leaf}\}\!\}, \{\!\{\mathsf{addr}\}\!\}, \{\!\{L\}\!\})$;	For each $\tau \in [0 : 2\mathsf{B})$, garble the GCSw_τ circuit (whose functionality is defined on the left);
2. For $b \in \{0,1\}$, $\{\!\{K_b, \_\}\!\} \leftarrow \mathsf{GStack}_b.\mathsf{Pop}(\{\!\{\beta_b\}\!\})$; *// here _ means ignore the last* 2λ *bits*	
3. Call $\{\!\{K = K_0 \oplus K_1\}\!\} \leftarrow \mathsf{GXOR}_\tau(\{\!\{K_0\}\!\}, \{\!\{K_1\}\!\})$;	For each $\tau \in [0 : 2\mathsf{B})$, garble the GXOR_τ circuit (which computes bitwise XOR);
4. Re-garble $\{\!\{\mathsf{leaf}', \mathsf{addr}', L'\}\!\}$ under language K using the given $\{\!\{K\}\!\}$.	For each $\tau \in [0 : 2\mathsf{B})$, create the re-garbling circuits;
• $\mathsf{Finalize}(\{\!\{1\}\!\})$:	
1. If $\mathbf{InitL} \neq \emptyset$, call: – $\{\!\{1_L, 1_R\}\!\} \leftarrow \mathsf{Dec}_{\{\!\{1\}\!\}}(\mathsf{ct}_\tau)$; – $\{\!\{\_, K'_0\}\!\} \leftarrow \mathsf{GStack}_0.\mathsf{Pop}(\{\!\{1_L\}\!\})$; – $\{\!\{\_, K'_1\}\!\} \leftarrow \mathsf{GStack}_1.\mathsf{Pop}(\{\!\{1_R\}\!\})$; – $\{\!\{1'_L\}\!\} := K'_0$, i.e., re-garble $\{\!\{1\}\!\}$ under K'_0; – $\{\!\{1'_R\}\!\} := K'_1$, i.e., re-garble $\{\!\{1\}\!\}$ under K'_1; and output $\{\!\{1'_L, 1'_R\}\!\}$. 2. Else, call – $\{\!\{\mathsf{st}_0\}\!\} \leftarrow \mathsf{GStack}_0.\mathsf{Finalize}(\{\!\{1\}\!\})$, – $\{\!\{\mathsf{st}_1\}\!\} \leftarrow \mathsf{GStack}_1.\mathsf{Finalize}(\{\!\{1\}\!\})$, and output $\{\!\{\mathsf{st}_0, \mathsf{st}_1\}\!\}$.	If $\mathbf{InitL} \neq \emptyset$, then, for each $\tau \in [0 : 2\mathsf{B}]$, create the ciphertext $\mathsf{ct}_\tau = \mathsf{Enc}_{\{\!\{1\}\!\}_\tau}(\{\!\{1_L, 1_R\}\!\}_\tau)$, and create the re-garbling circuits.

Figure 13: $\mathsf{GSwitch}$ algorithm from OWF.

```
Evaluator

• $\mathsf{Init}(\{\!\{\mathbf{DB}\}\!\})$:
1. for $i \in [0:m)$: $\mathsf{GOTStack}_i.\mathsf{Push}(\{\!\{\mathbf{DB}[i]\}\!\})$;
• $\mathsf{Read}(\{\!\{\mathsf{idx}^E\}\!\})$:
1. let $\{\!\{L^0\}\!\} = \{\!\{\mathbf{OutL}[\tau]\}\!\}$;
2. let $\{\!\{\mathsf{leaf}^0\}\!\} = \{\!\{\mathsf{idx}\}\!\}$, $\{\!\{1^0\}\!\} = \{\!\{1\}\!\}_\tau$;
3. let $V =$ root and $\ell_{\max} = \log_2 m$;
4. for $\ell \in [0:\ell_{\max})$:
   – $\{\!\{\mathsf{leaf}^{\ell+1}\}\!\}, \{\!\{1^{\ell+1}\}\!\}, \{\!\{L^{\ell+1}\}\!\}$
        $\leftarrow \mathsf{GSwitch}^V.\mathsf{Switch}(\{\!\{\mathsf{leaf}^\ell\}\!\}, \{\!\{1^\ell\}\!\}, \{\!\{L^\ell\}\!\})$
   – $V \leftarrow V.\mathsf{child}[\mathsf{idx}[\ell]]$;
   // $V$ is now the $\mathsf{idx}$-th leaf
5. $\{\!\{\mathsf{rdata}\}\!\} \leftarrow \mathsf{GOTStack}_{\mathsf{idx}}.\mathsf{Pop}(\{\!\{1^{\ell_{\max}}\}\!\})$;
6. re-garble $\{\!\{\mathsf{rdata}\}\!\}$ under $L^{\ell_{\max}}$ and then output the result;
• $\mathsf{Finalize}(\{\!\{1\}\!\})$:
1. $\mathsf{RecFinalize}(\text{root}, \{\!\{1\}\!\})$ defined below;
2. output $\{\!\{\mathbf{DB'}\}\!\} = \{\{\!\{D_i\}\!\}\}_{i \in [0:m)}$.
   Subroutine $\underline{\mathsf{RecFinalize}(V, \{\!\{1\}\!\})}$
a) if $V$ is a leaf, let $i$ be its index:
      $\{\!\{D_i\}\!\} \leftarrow \mathsf{GOTStack}_i.\mathsf{Finalize}(\{\!\{1\}\!\})$
   and return; else continue with the following;
b) $\{\!\{1_L, 1_R\}\!\} \leftarrow \mathsf{GSwitch}^V.\mathsf{Finalize}(\{\!\{1\}\!\})$
c) let $U_L$ and $U_R$ be $V$'s left and right children;
d) $\mathsf{RecFinalize}(U_L, \{\!\{1_L\}\!\})$; $\mathsf{RecFinalize}(U_R, \{\!\{1_R\}\!\})$.

Garbler

for $i \in [0:m)$:                // see narrative
   call $\mathsf{GOTStack}_i.\mathsf{Garble}$;

for $\tau \in [0:m)$:
   create the garbling $\{\!\{\mathbf{OutL}[\tau]\}\!\}$;
   create the garbling $\{\!\{1\}\!\}_\tau$;

for each tree node $V$:     // see narrative
   call $\mathsf{GSwitch}^V.\mathsf{Garble}$;
```

Figure 14: GOTM algorithm from OWF.

B.2 Garbled Memory

Given the above building blocks, we start with the non-recursive garbled memory, and then we describe the full construction. In the non-recursive garbled memory, we use n and w to denote the number of elements and the bit-width of each element, and then in the recursive one, we use N and W to denote the number of memory words and the bit-width of each word.

Non-recursive garbled memory (NRGRAM). The non-recursive garbled memory is the main difference between variants using Free-XOR and Yao's garbling. Recall that in the construction of Section 6, we instantiated a Bucket ORAM tree, each tree node is associated with a bucket and a switch; each operation on the garbled memory visits a root-to-leaf path, each bucket on the path is visited and outputs a garbled data, but only one garbled data is real while others are garbled zeros, so the garbled data are aggregated by XORing the garbled data, and finally the aggregation is "re-garbled" into the desired garbling using the languages forwarded through the switches on the path. Using Yao's garbling, we cannot XOR the garbled data in the same way since the encodings are no longer linearly homomorphic. One naive solution is to forward $\log n$ global-time languages to $\log n$ buckets through the switches, each bucket consumes one language to re-garble the data, and finally use a garbled circuit to compute the result of XOR. But this approach incurs extra logarithmic factor on cost.

Instead, we compute a partial aggregation immediately (using XOR) when we obtained the garbled data from the bucket, and then we forward the XORed element down the path using our switches. That is, at each node, we firstly read the desired address from the bucket, then XOR the newly read data with the data forwarded from the parent node, and finally forward the XORed data to the corresponding (either left or right) child using the switch. On the same path, the switches also forwards the global-time language (in the garbled form), so that at the leaf level we re-garble the aggregated data under the global-time language.

The modified procedure is given in Figure 13, where the other functions ($\mathsf{Add}, \mathsf{RecFinalize}$) are identical to that of Figure 8.

- Stash costs $O(\lambda n(w + \log n))$.
- For each tree node, a bucket and a switch cost

$$O\left(\mathsf{B} \cdot \lambda^2 \cdot (\log n \cdot \log^3 \mathsf{B} + w \log^2 \mathsf{B} + \log^4 \mathsf{B})\right) + O\left(\mathsf{B}\lambda \cdot (\log n + \lambda w) \cdot \log \mathsf{B}\right)$$
$$= O\left(\mathsf{B} \cdot \lambda^2 \cdot (\log n \cdot \log^3 \mathsf{B} + w \log^2 \mathsf{B} + \log^4 \mathsf{B})\right).$$

- For every n operations, we instantiate $O((n/B)\log n)$ tree nodes, which costs

$$O\left(n \log n \cdot \lambda^2 \cdot (\log n \cdot \log^3 \mathsf{B} + w \log^2 \mathsf{B} + \log^4 \mathsf{B})\right).$$

- XOR and re-garbling cost $O(\lambda^2 w \log n)$ per operation. That is, $O(\lambda^2 wn \log n)$ per n operations.
- For every n operations, for each level ℓ, the level rebuild is performed $O(n/(\mathsf{B}2^\ell))$ times. Hence all levels takes $O(\lambda n \log n \cdot (\log n + w))$ per n operations.

We conclude that the tree nodes is dominating, and for $T > n$ the cost is $O\big(n \log n \cdot \lambda^2 \cdot (\log n \cdot \log^3 \mathsf{B} + w \log^2 \mathsf{B} + \log^4 \mathsf{B})\big)$.

Evaluator	**Garbler**
$\underline{\mathsf{ReadRm}\left(\{\!\{\mathsf{addr}, \mathsf{leaf}^E\}\!\}\right):}$ • if $t = 0$, then for every tree node V, call $\mathsf{GBkt}^{V,0}.\mathsf{Init}(\{\!\{\mathsf{bkt}_V^{\emptyset}\}\!\})$;	for every tree node V, let $\mathsf{bkt}_V^{\emptyset}$ be an array of 0s of appropriate length, create garbled state $\{\!\{\mathsf{bkt}_V^{\emptyset}\}\!\}$;
• $\_, \{\!\{\mathsf{rdata}_{-1}\}\!\} \leftarrow \mathsf{GStash}^t.\mathsf{Read}(\{\!\{\mathsf{addr}\}\!\})$;	
• $\{\!\{\mathsf{rdata}_{-1}, L\}\!\} := (\{\!\{\mathsf{rdata}_{-1}\}\!\}, \{\!\{\mathbf{L}^*[t]\}\!\})$; • For each node V in the tree from the root to leaf, – $\{\!\{\mathsf{rdata}'_\ell\}\!\} \leftarrow \mathsf{GBkt}^{V,t}.\mathsf{Read}(\{\!\{\mathsf{addr}\}\!\})$ where ℓ denotes the level of V;	for $t \in [0:T)$, let $\mathbf{L}^*[t] = \mathbf{OutL}[t]$; create garblings $\{\!\{\mathbf{L}^*\}\!\} := \{\{\!\{\mathbf{L}^*[t]\}\!\}\}_{t\in[0:T)}$ under the input languages of $\mathsf{GSwitch}^{\mathrm{root},t}.\mathbf{InL}$ and $\mathsf{GBkt}^{\mathrm{root},t}.\mathbf{InL}$;
– $\{\!\{\mathsf{rdata}_\ell\}\!\} \leftarrow \mathsf{GXOR}^{V,t}(\{\!\{\mathsf{rdata}_{\ell-1}\}\!\}, \{\!\{\mathsf{rdata}'_\ell\}\!\})$;	call $\mathsf{GBkt}.\mathsf{Garble}$ for all GBkt instances;
– If V is not a leaf: let $\{\!\{\mathsf{leaf}\}\!\}, \{\!\{\mathsf{addr}\}\!\}, \{\!\{\mathsf{rdata}_\ell, L\}\!\} \leftarrow \mathsf{GSwitch}^{V,t}.\mathsf{Switch}(\{\!\{\mathsf{leaf}\}\!\}, \{\!\{\mathsf{addr}\}\!\}, \{\!\{\mathsf{rdata}_\ell, L\}\!\})$ (that is, re-garble and forward the message);	call $\mathsf{GSwitch}.\mathsf{Garble}$ for all $\mathsf{GSwitch}$ instances;
• Let $\{\!\{\mathsf{rdata}\}\!\}$ be the re-garbling of $\{\!\{\mathsf{rdata}_{\ell_{\max}}\}\!\}$ under L and then output $\{\!\{\mathsf{rdata}\}\!\}$.	see $\mathsf{GSwitch}$ for the re-garbling.

Figure 15: Non-Recursive Garbled RAM (NRGRAM) construction from OWF (see Figure 8 for functionalities Add and RecFinalize).

Full construction: garbled memory (GRAM). The full construction follows identically the procedures of Section 7: given N data blocks, each consists of W bits, we store the N blocks in a non-recursive GRAM, and then recursively store the position of each block in $D = \log N$ instances of non-recursive GRAM, where the number of blocks reduces by a factor of two in each recursion. Notice that the size of each element in the recursion is $\log N$ bits except for the outer-most non-recursive GRAM. Hence, per N accesses, the cost is

$$\begin{aligned}
&O\big(N\log N\cdot\lambda^2\cdot(\log N\cdot\log^3\mathsf{B} + W\log^2\mathsf{B} + \log^4\mathsf{B})\big)\\
+&\sum_{i=1}^{D} O\big(N\log\frac{N}{2^i}\cdot\lambda^2\cdot(\log\frac{N}{2^i}\cdot\log^3\mathsf{B} + \log N\log^2\mathsf{B} + \log^4\mathsf{B})\big)\\
=&O\big(N\log N\cdot\lambda^2\cdot(\log^2 N\cdot\log^3\mathsf{B} + W\log^2\mathsf{B} + \log N\log^4\mathsf{B})\big).
\end{aligned}$$

Following Section 7 and hiding the statistical failure probability in the $\widetilde{O}(\cdot)$ notation, we simplify the expression to

$$\widetilde{O}\big(T\cdot\lambda^2\cdot(\log^3 N + W\log N)\big)$$

per $T \geq N$ accesses. This takes an extra λ factor compared to Section 7.

Analysis of correctness and security.

- Non-recursive garbled memory (NRGRAM). The analysis is almost identical to that of Theorem 6.1, but the hybrids Hyb_2 and Hyb_t^* are modified as follows due to the construction.

Recall that we take a different approach to forward and aggregate the read data (i.e., rdata in Figure 15). Hence in Hyb_2, we sample the garbling $\{\!\{\mathsf{rdata}\}\!\}_t$ for each $t \in [0, T)$ uniformly at random (and skip all other garblings, including all $\{\!\{\mathsf{rdata}_\ell\}\!\}_t$ for all ℓ). Next for each Hyb_t^*, the building blocks are simulated in the ordering below.

1. Simulate the re-garbling of $\{\!\{\mathsf{rdata}_{\ell_{\max}}\}\!\}$ (under language L);
2. for each node from $\ell_{\max}$ down to 0, sequentially simulate the switch, the XOR circuit, and then the bucket;
3. simulate the stash, and then simulate the rebuild circuit (which is identical to the proof of Theorem 6.1).

This sketches the simulator of NRGRAM.

- Full garbled memory (GRAM). The analysis is exactly identical to Section 7.3.

The full construction is concluded as below.

Theorem B.1 (Garble RAM from one-way functions)**.** *Assume the existence of one-way functions. Suppose that N, T, W are polynomially bounded in the security parameter λ, and moreover, we adopt $\mathsf{B} = \omega(\log \lambda)$ to instantiate the underlying NRGRAM instances. Then, the resulting GRAM scheme in this section respects Definition 1 and costs $\widetilde{O}\big(T \cdot \lambda^2 \cdot (\log^3 N + W \log N)\big)$ bits, where $\widetilde{O}(\cdot)$ hides $\mathsf{poly} \log \log \lambda$ factors.*

C More Detailed Comparison with Prior Work

C.1 Blackbox Garbled RAM Constructions

Garg et al. [GLO15] constructed the first blackbox garbled RAM from one-way functions. Their work mainly focused on feasibility, and did not care about the constant-exponent in the $\mathsf{poly}\log$ term. For a better comparison, we sketch their construction below to calculate the exact constant-exponent in their $\mathsf{poly}\log$ term. We refer the reader to the original paper [GLO15] for more details.

Construction of Garg et al. [GLO15]. At a high level, the construction begins from a RAM program already compiled with a statistical ORAM (e.g., with Circuit ORAM [WCS15, CS17]), such that the memory is partitioned into "levels." For each step, the program accesses a uniformly random location inside a pre-determined level. It is then sufficient to construct a garbled RAM scheme for programs that perform uniform memory accesses (over the whole memory) because we can naturally instantiate one garbled RAM for each level.

To do so, they construct a binary tree in which each leaf is associated with a memory index. For each computation time step t that accesses memory location i, the purpose of the tree is to send from the root to the i-th leaf the language of time t, so that the memory word at location i can then be *re-garbled using the given language*. Each internal node of the tree is provided with a sequence of garbled circuits. For a fixed circuit, we denote the previous or next circuit in the sequence as the *predecessor* or *successor*. At the evaluation, each memory access sends the language through the circuits along the corresponding root-to-leaf path, consuming at least one circuit for each node on the path. Because the memory accesses are uniform, the number of expected visits to any node is half the number of visits of its parent node. If this always holds, then we could wire every pair of circuits on the parent node to a pair of circuits on the left and right children nodes. Unfortunately, because the paths are uniformly sampled at evaluation, the number of visits to the left and right nodes can vary greatly.

Garg et al. addressed this challenge using two key insights. The first insight is to connect a circuit from a parent node to a *window* of circuits on its left and right children. Concretely, for some "window size" κ, each parent node circuit is connected to κ left-child and κ right-child circuits. Thus, it allows for some leeway at evaluation time. Then in the evaluation, the parent node routes its message to the next unvisited circuit of the left or right child depending on the path. As long as the next left-child or right-child circuit lies within the window, the routing can be performed as planned.

The second insight complements the first. In some cases, the window size may still not be enough if the numbers of routes to the left and right child vary widely. To this end, Garg et al. propose a "burning circuit", which allows a node to "burn" through its circuits such that the desired child circuit becomes part of the window for subsequent accesses. For example, suppose a parent node u is going to visit left child v, but the right child w has been visited κ times more than v. The parent circuit C_u^i first sends a message to the next left child $C_v^{i'}$ even though i' is not part of the window of i, and the message is also encoded with the targeted window t' Given t', child $C_v^{i'}$ checks and knows it is behind the window, so $C_v^{i'}$ chooses to burn circuits $C_v^{i'}, C_v^{i'+1}, \ldots, C_v^{t'-1}$, and they forward the message through the circuits to successor circuit $C_v^{t'}$. The circuit $C_v^{t'}$ can now continue the routing to its child, and the later parent circuits $C_u^{i'+1}, C_u^{i'+2}, \ldots$ have their next left children $C_v^{t'+1}, C_v^{t'+2}, \ldots$ in their windows.

Finally, to realize the above insights, each circuit is connected to its successor, and the successor receives some tracking information. Namely, the tracking information includes the numbers of visits to the left and right children, and the languages of the next left- and right-child circuits (the language is needed to initiate burning). Moreover for each parent circuit i, the window is parameterized to the children circuits in the range $[(1/2+\epsilon)\cdot i, (1/2+\epsilon)\cdot i+\kappa]$, where ϵ and κ are chosen properly for efficiency and security.

Analysis and cost. In the above construction, the cost and failure probability depend on parameters κ and ϵ. Notice that the construction will still fail when one child is visited too many times, e.g., if a parent visits the right child consecutively for 4κ times, then the parent runs out of hardwired language to the next right circuit. For the same reason, choosing $\epsilon > 0$ is necessary—otherwise, the difference in the number of visits eventually accumulates and leads to failure. Hence, ϵ is chosen to $1/\log n$ where n is the number of memory words, and this incurs an $O(\log n)$ factor on the total number of circuits over the whole tree [GLO15, Section 5.1.1]. Next, for any $\epsilon > 0$, the failure probability is analyzed and bounded by $\exp[-(\epsilon^2\kappa+\epsilon(\kappa-1))]$ in Garg et al. [GLO15, Lemma 5.2]. Hence, choosing $\kappa = \omega(\log^2 n)$ is necessary to achieve a negligible failure probability in n.

Together, there are $O(n \log n)$ circuits per n accesses, each circuit hardwires $O(\kappa)$ languages, each language must be able to encode at least a w-bit memory word and thus is $O(\lambda w)$ bits. Thus, the cost is $O(n \log n \cdot \kappa\lambda^2 w)$, where the second λ factor comes from garbling those circuits that hardwire languages. Recall that this is only the garbled RAM for uniformly random accesses. Given a memory array consisting of N blocks, each block is W bits, we compile it with Circuit ORAM, which incurs $O(\log N)$ accesses to W-bit blocks, and $O(\log^2 N)$ accesses to $(\log N)$-bit words (of metadata). Hence, the per-access cost with Circuit ORAM is

$$O\left((\log^2 N \cdot W + \log^4 N)\cdot \kappa\lambda^2\right).$$

Plugging in $\kappa = \widetilde{O}(\log^2 N)$, the cost is $\widetilde{O}\left((W + \log^2 N)\cdot \log^4 N \cdot \lambda^2\right)$ per access, supposing that N and the total number of accesses are bounded by a polynomial of λ.

Garbled Parallel RAM programs from one-way functions. Lu and Ostrovsky [LO17] extend the above construction to garble Parallel RAM programs where M processors concurrently access N memory words. When it comes to parallel accesses, the above construction no longer works efficiently because each circuit on tree nodes sends only one language (while there are M concurrent accesses). To resolve this problem, Lu and Ostrovsky proposed the technique called "wide circuits" for those nodes near the root. For all internal nodes such that is $i < \log_{4/3} M$ distance from the root, their associated circuits are *all activated* and send messages in every access, and the circuits are "wide" that take as input several languages and then send them to both the left and right children, where the "width" is decreased with respect to increasing level i to reduce the cost.[9] For each internal node such that is $i > \log_{4/3} M$ distance from the root, the circuits are connected identically to those of the sequential garbled RAM (summarized above). Finally, at level $i = \log_{4/3} M$, the circuits receive as input at most $O(\log^4 N)$ languages with overwhelming probability, and hence the circuits just send the languages sequentially to their children. This garbled PRAM scheme does not improve the efficiency of garbled RAM, and hence it is less relevant to our scheme.

C.2 Garbled RAM from Non-Blackbox One-Way Functions

For completeness, we summarize the non-blackbox constructions in this subsection. Garg et al. constructed the first garbled RAM from one-way functions in a non-blackbox manner [GLOS15], i.e., the construction involves garbling the circuits that evaluate PRF. Then, Hazay and Lilintal improved the efficiency [HL20]. These works also cared mostly about feasibility, and did not care about the exact constant-exponent in the $\mathsf{poly}\log$ term. Hazay and Lilintal calculated exactly how large the $\mathsf{poly}\log$ term is but only for the number of garbled evaluations of the PRF, which is the most expensive part of non-blackbox constructions. Below, we sketch their schemes at a high level by listing the data and circuits, so we can derive a more exact expression of their cost. We refer the readers to the original papers for more details. [GLOS15, HL20].

Given n memory words, a binary tree of n leaves is constructed, each internal node is associated with a ciphertext encrypting two PRF keys (there is no circuit associated with any node), and for each internal node, the two PRF keys are the encryption keys of the ciphertexts of the left and right children correspondingly. That is, for a node x where the parent node is w, the left and right children are y, z, the ciphertext associated to x is $f_{k_w}(k_y \| k_z)$ (and those ciphertexts to y and z are $f_{k_y}(\dots)$ and $f_{k_z}(\dots)$).

The tree of encrypted keys is designed to securely evaluate the following garbled circuits. For each memory access t and for each level i of the tree, a garbled circuit C_i^t is instantiated:

1. Circuits $C_0^t, \dots, C_{\log n}^t$ sequentially traverse a path from the root to a leaf in the tree of ciphertexts (the memory word to be read is stored in some node of the path).

2. Each circuit C_i^t takes as input the two keys k_l, k_r from the node at level i on the path (we will soon explain how to obtain the keys from the ciphertext of the node).

3. Circuit C_i^t is hardwired with keys k_{i-1}^t, k_i^t and the garbling language of C_{i+1}^t, where k_i^t is a fresh and independent PRF key for all t, i, and k_{i-1}^t is the same key that is hardwired in circuit C_{i-1}^t.

[9] Concretely, for $i < \log_{4/3} M$, the level i circuit takes as input M_i languages concurrently and then sends M_{i+1} languages to each of left and right children, where level 0 is the root, $M_0 := M$, and M_{i+1} is defined iteratively by $M_{i+1} := (1/2 + 1/\log N)M_i + \log^3 N$ [LO17, Figure 1].

4. Choose $k^* \in \{k_l, k_r\}$ that is encrypting the next ciphertext on level $i+1$ in the path (as k_l or k_r denotes the left or right key), evaluate PRF using key k^*, and then XOR the result with the garbling language of C^t_{i+1}. The result of XOR is called the "translate".

5. Replace k^* with k^t_i in the pair (k_l, k_r), encrypt the new pair with k^t_{i-1} by evaluating PRF on every bit of the pair using key k^t_{i-1}.

6. Both the translate and the encryption are output by C^t_i.

7. To obtain the input of the next garbled circuit C^t_{i+1}, the evaluator of GRAM performs XOR on the translate and the ciphertext on level $i+1$ in the path (which translates the ciphertext into the garbling of the keys, as desired in Step 2). The evaluator also puts the encryption to the level i node on the path.

The above shows only the encryption and activation of keys, but auxiliary control and data can be incorporated (e.g., additionally store the data of a tree-based ORAM in the tree of encrypted keys [HL20]). Notice that each garbled circuit evaluates PRF to encrypt the keys and data, but *garbling a PRF circuit* is non-blackbox and costly. Suppose the key of the PRF is λ bits, let S_λ be the circuit size of the PRF such that outputs λ bits, and suppose the ORAM tree stores and processes w bits per node (e.g., Circuit ORAM). The hardwired language is $O(\lambda(\lambda + w))$ bits, there are $O(\lambda + w)$ PRF evaluations in the garbled circuit, and hence each garbled circuit costs $O(\lambda^2(\lambda + w) + \lambda S_\lambda(\lambda + w))$ bits. Circuit ORAM compiles each memory access into accessing (a) $O(\log N)$ tree nodes each of $O(W)$ bits and (b) $O(\log^2 N)$ tree nodes each of $O(\log N)$ bits. Hence, as $S_\lambda \geq \lambda$, the total cost is

$$O(\log N \cdot \lambda S_\lambda(\lambda + W) + \log^2 N \cdot \lambda S_\lambda(\lambda + \log N)) = O\left(\log N \cdot \lambda S_\lambda(W + \lambda \log N + \log^2 N)\right)$$

bits per access.

C.3 Early Scheme Based on Circular Security

In their pioneering work, Lu and Ostrovsky proposed the first garbled RAM scheme [LO13]. Although the scheme requires a complex circular use of garbled circuits and PRFs [GHL+14], we briefly show the idea and cost of the scheme.

Very roughly, the scheme of Lu and Ostrovsky constructs a pair of circuits (C_{CPU}, C_{ORAM}) for each logical memory access. C_{CPU} performs one step of computation and outputs one logical memory address to be accessed. C_{ORAM} takes in the logical address and outputs a sequence of $O(\log N)$ garbled circuits (each performs one physical access to emulate the ORAM). Then, each pair (C_{CPU}, C_{ORAM}) is garbled and provided to the evaluator. Finally, the evaluator evaluates garbled C_{CPU}, garbled C_{ORAM}, obtains the sequence of garbled circuits, and then evaluates the sequence (which completes one logical access).

To create the garbled circuits, C_{ORAM} is hardwired with a secret key (this is also why circular security is needed). Recall that garbling a constant fan-in Boolean gate takes circuit size $O(S_\lambda)$ to evaluate PRF and encrypt all potential outputs. Because the ORAM accesses $\tilde{O}(w \log n)$ bits per physical access, garbling each circuit takes circuit size $\tilde{O}(w S_\lambda \log n)$. C_{ORAM} outputs $O(\log n)$ garbled circuits, and thus it takes circuit size $\tilde{O}(w S_\lambda \log^2 n)$. The garbling of C_{ORAM} incurs another λ factor, and hence it takes $\tilde{O}(w \lambda S_\lambda \log^2 n)$ bits per memory access.