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

Andrew Park[1], Wei-Kai Lin**[2][0000−0001−6012−7124], and Elaine Shi[1]

[1] Carnegie Mellon University
ahp2@andrew.cmu.edu, runting@cs.cmu.edu
[2] Northeastern University we.lin@northeastern.edu

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. (Eurocrypt '22). 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 of 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 [39, 40], 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.

* Author ordering is randomized. The full version of the paper can be accessed at https://eprint.iacr.org/2022/191.

** The work was done while the author was a postdoctoral researcher at CMU.

The evaluator learns nothing about the garbled inputs or outputs. Subsequently, numerous works have focused on making garbled circuits increasingly more practical [1,10,21–26,32,39,40,42]. 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 [28] 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 N denotes the space of the RAM. Throughout the paper, we often use the metric "amortized cost per memory access" to characterize the performance of a garbled RAM scheme, which is the number of bits that must be communicated per memory access. Since the original work of Lu and Ostrovsky [28], a line of works [14,17,24,29] have focused on improving garbled RAM constructions.

With the exception of the most recent work by Heath et al. [24], prior works on garbled RAM [14, 17, 29] 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. [24] 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 [10, 26] and subsequent improvements [25, 32, 42].

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 [37] 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. [24]. 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. [24], 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 compar-

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 G and H of the online full version [30] for details.

	Assumption	Cost per access	Blackbox
Lu and Ostrovsky [28]	Circular GC[a]	$\widetilde{O}(\lambda S_\lambda W \log^2 N)$	No
Hazay and Lilintal [20]	OWF	$O(\lambda S_\lambda \cdot (W \log N + \lambda \log^2 N + \log^3 N)$	No
Garg et al. [14]	OWF	$\widetilde{O}\left(\lambda^2 \cdot (W \log^4 N + \log^6 N)\right)$	Yes
Heath et al. [24]	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
	CCR hashes[c]	$O\left(\lambda \mathsf{B} \cdot (W \log N + \log^3 N)\right)$	Yes
	OWF	$\widetilde{O}\left(\lambda^2 \cdot (W \log N + \log^3 N)\right)$	Yes

a. Circularly secure garbled circuit, see [17].
b. This is not documented in their paper, but it is a standard method to tweak their scheme.
c. Our practically efficient scheme, where B is the statistical security parameter.

ison with Heath et al. [24], we save almost a logarithmic factor. Our scheme makes the same assumptions as Heath et al. [24] as well as the standard literature on efficient garbled circuits [10, 25, 26, 32, 42], 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 (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. 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 [18, 19, 27]. 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[3]. Although computationally secure ORAMs [2,31] 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 garble the PRF employed by the ORAM. Third, as mentioned, even when allowing interactions, we do not know

[3] 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.

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 (Garbled RAM from one-way functions). *Assume the existence of one-way functions. There exists a garbled RAM scheme that achieves $\widetilde{O}\big(\lambda^2 \cdot (\log^3 N + W \log N)\big)$ 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 $\widetilde{O}\big(\lambda^2 \cdot W \log N)\big)$ amortized cost per memory access.

Compared to prior works. Table 1 compares our asymptotical result with prior garbled RAM works. The ealier works (e.g., [14,17,28]) used ORAM as a black-box and did not care about how large the $\mathsf{poly}\log$ is. Both Heath et al. [24] and our work observe that to optimize the $\mathsf{poly}\log$ factors, we need to open up the underlying ORAM, and tailor the ORAM's design specifically for garbled RAM. In our paper, a key observation is that the more uncertainty there is regarding which address will be accessed, the more overhead we need to pay to account for the uncertainty. Therefore, one of our main techniques is to localize the uncertainty (of which address is accessed) to polylogarithmically sized regions.

Besides those listed in the table, Gentry et al. [17] also propose a garbled RAM scheme from one-way function and identity-based encryption with polylogarithmic 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. [14]. The table also did not include reusable Garbled RAM [4,5,9] which are based on indistinguishability-based obfuscation (iO). Known reusable garbled RAM constructions can compress the total communication but they do not save the evaluator's runtime.

Concrete performance. In additional to our main results, we explore the concrete performance. In Appendix A of the online full version [30], we suggest several practical optimizations to our garbled RAM scheme described in Theorem 1. Our practically efficient scheme eliminates constant and $\mathsf{poly}\log\log$ factors while introducing a statistical security parameter, as shown in Table 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 [24] at about $N = 2^{13}$ memory size.

2 Technical Roadmap

2.1 Background

Encodings. We will use the following forms of encodings.

- *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 [26]. 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* [24] 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 [14,17,24,28,29], 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. [24], 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. [24] proposed an elegant idea that leverages two garbled stacks [24, 38, 42] 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 provide. 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 [24, 38, 42], 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. [24] is inefficient. At a very high level, Heath et al. [24] 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. [24] 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[4] 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. [24], earlier works on garbled RAM [14,17,28,29] 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 [7, 37]; 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 polylogarithmic overhead, and the two sources of overheads are multiplied. Heath et al. [24] 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

[4] 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.

a state-of-the-art *statistically* secure ORAM[5] such as Circuit ORAM [7,37], 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 [7,37]. The full ORAM scheme involves creating logarithmically many such trees through a standard recursion technique [33,35]. 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. [16].

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.

[5] 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.

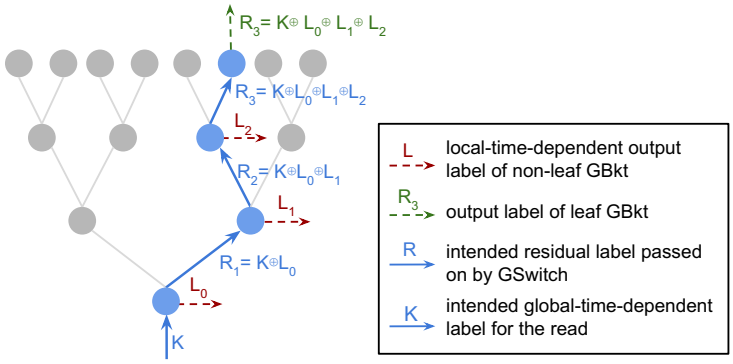


Fig. 1: XOR trick.

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.

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 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.

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 [13].

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 [33], 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 [7, 33, 34, 37]. 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 [6, 8, 11] 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 [18, 19]. 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

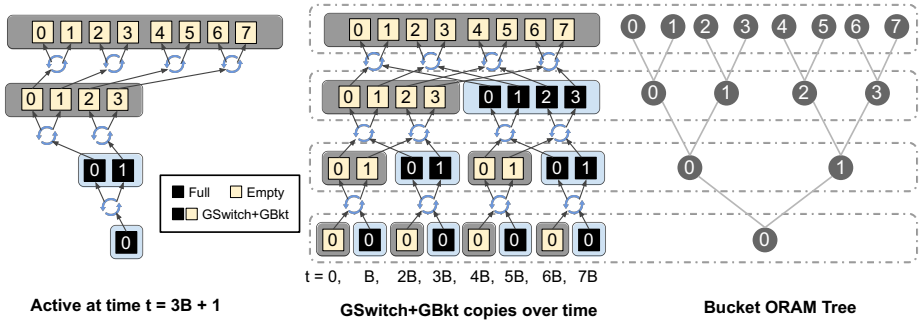


Fig. 2: Each node at level ℓ has $T/(\mathsf{B} \cdot 2^\ell)$ copies of $\mathsf{GSwitch} + \mathsf{GBkt}$, and at time t, the $\lfloor t/(\mathsf{B} \cdot 2^\ell) \rfloor$-th copy is active. The numbers show which copies of garbled circuitry correspond to which tree node in the same level.

levels $0, \ldots, \ell - 1$ into level ℓ, emptying the levels $0, \ldots, \ell - 1$ in the process. Further, the rebuild process must respect the position identifier each block has chosen. The Bucket ORAM work [13] 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 [13] for details. At a high level, the Bucket ORAM work suggested that this rebuild can be accomplished through a sequence of $\mathsf{MergeSplit}$ operations. In each $\mathsf{MergeSplit}$ 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 $\mathsf{MergeSplit}$ operation essentially relies on sorting of objects with 1-bit keys, i.e., compaction [2]. Indeed, if we use a linear-sized compaction circuit to realize each $\mathsf{MergeSplit}$, 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 $\mathsf{MergeSplit}$, 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 $\mathsf{GSwitch}$. 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 $\mathsf{GSwitch}$ 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.

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 **DB**, input and output labels denoted **InL** and **OutL**, and outputs the garbled circuit GC and initial garbled memory Gmem. Specifically, **InL** and **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 Finalize in the intended time step t^* and not any other time step. The 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 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 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, and we give efficient instantiations partly relying on a building block called an expiring vault (see Appendix D.1 of the online full version [30]).

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. [24]. Similarly, for other seemingly standard building blocks such as garbled stack, we also have to construct our own variants and prove them secure.

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. [24]. 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., $(\llbracket x \rrbracket, \llbracket y \rrbracket) \to \llbracket x \oplus y \rrbracket$;
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., $\left(\{\!\{b^E\}\!\}, \llbracket y \rrbracket\right) \to \llbracket b \cdot y \rrbracket$ 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. [24] 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 Definitions: Garbled Data Structure

Recall that in Section 2.1, we defined two types of encodings called *sharings* and *garblings* for garbled circuits. We refer the reader to Appendix B of the online full version [30] for a more detailed review of garbled circuits. We now proceed to define garbled data structures.

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 evaluate 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 **InL** and **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 $1 \le t^* \le t_{\max}$, 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:

– $\mathsf{Gmem}, \mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{In}\ , \mathbf{LOutDBu})$;

– 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 below.

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* $1 \le t^* \le t_{\max}$*, 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, $\mathsf{GC} \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{sk}, \mathsf{params}, \mathbf{DB}, \mathbf{InL}, \mathbf{OutL})$;
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 $\mathbf{DB}[\neg S]$, output labels $\mathbf{utLO}$.

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 $\mathbf{DB}[\neg S]$;
3. Run the ideal functionality using the given $\mathbf{In}$, 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 [1, 10, 25, 26, 32, 39, 40, 42], including the techniques needed from Heath et al. [24] 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 Appendix B.1 of the online full version [30], 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 [24, 42], 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. In this paper, we measure cost by the size of the garbled program, in terms of the number of bits. We often use the metric "cost per access" where we amortize the total cost over the number of memory accesses.

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 4.2 — 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 [1, 10, 21–26, 32, 39, 40, 42] 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**,

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

where GC is the honest garbling of the circuit C using randomly generated input labels as well as **OutL**, and $\stackrel{c}{\equiv}$ 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.

3.1 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 $\widetilde{\mathsf{outp}} \leftarrow \mathsf{GDataStruct}.\mathsf{Func}_i(\widetilde{\mathsf{inp}})$, 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:

$$\begin{array}{l}\mathsf{GDataStruct}_0.\mathsf{Func}(\{\!\{x\}\!\}) : \\ \quad \{\!\{y\}\!\} \leftarrow \mathsf{GDataStruct}_1.\mathsf{Foo}(\{\!\{x\}\!\}); \\ \quad \{\!\{z\}\!\} \leftarrow \mathsf{GDataStruct}_2.\mathsf{Bar}(\{\!\{y\}\!\}); \\ \quad \text{output } \{\!\{z\}\!\};\end{array}$$

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 for Garbled Memory

4.1 Stack (GStack)

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 input/output garbling labels denoted $\mathbf{InL}$ and $\mathbf{OutL}$ 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 $\mathbf{OutL}$.
- $\{\!\{\mathsf{ucnt}\}\!\} \leftarrow \mathsf{Finalize}^{\mathsf{GC}}(\mathsf{Gmem}, \{\!\{1\}\!\})$: 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 $\mathbf{OutL}$.

Construction Although efficient garbled stacks have been proposed in earlier works [24, 38, 41, 42], we need a variant that supports dynamic finalization. To support this new feature, we propose a new abstraction called a garbled vault denoted GVault in Appendix D.1 of the online full version [30]. We use GVault to construct a new garbled stack with dynamic finalization in Appendix D.2 of the online full version [30].

4.2 Switch (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.[6] 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 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. [24] 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 [13], and since its choice is random, it is safe to reveal leaf.

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

[6] Using switches of arity-2 is the most efficient with our current techniques.

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 $\mathbf{InL} \quad \mathbf{InitL}, \mathbf{ReqL}, \mathbf{CtrlL}$ 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.

Construction Although Heath et al. [24] describe a garbled switch scheme for constructing an access-revealing garbled one-time memory, again we need a new

Evaluator	**Garbler**
– $\mathsf{Init}(\{\!\{\mathsf{st}_\emptyset^E\}\!\})$: *// called when* $\mathbf{InitL} \neq \emptyset$ 1. parse $\{\!\{\mathsf{st}_\emptyset^E\}\!\} := \{\!\{\beta_{b,i}\}\!\}_{b\in\{0,1\},i\in[0:2\mathsf{B})}$; 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 $\{\!\{\beta_0 = \mathsf{leaf}[0]\}\!\}, \{\!\{\beta_1 = 1-\beta_0\}\!\}, \{\!\{\mathsf{leaf}' = \mathsf{leaf}[1:]\}\!\}, \{\!\{\mathsf{addr}' = \mathsf{addr}\}\!\}, L' = L \oplus \mathbf{RdL}[\tau] \leftarrow \mathsf{GCSw}_\tau(\{\!\{\mathsf{leaf}\}\!\}, \{\!\{\mathsf{addr}\}\!\}, L, \mathbf{RdL}[\tau])$;	For each $\tau \in [0 : 2\mathsf{B})$, create the sharing $[\![\mathbf{RdL}[\tau]]\!]$, and 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. Output $(\{\!\{\mathsf{leaf}'\}\!\}, \{\!\{\mathsf{addr}'\}\!\}, L') \oplus K_0 \oplus K_1 \oplus \mathsf{TrL}_\tau$.	For each $\tau \in [0 : 2\mathsf{B})$, compute the translation label $\mathsf{TrL}_\tau := \mathsf{Lbl}([\![K_0]\!]_\tau) \oplus \mathsf{Lbl}([\![K_1]\!]_\tau) \oplus \mathsf{Lbl}(\{\!\{\mathsf{leaf}', \mathsf{addr}'\}\!\}_\tau, [\![L']\!]_\tau)$;
– $\mathsf{Finalize}(\{\!\{1\}\!\})$: 1. If $\mathbf{InitL} \neq \emptyset$, call: • $\{\!\{1_L, 1_R\}\!\} \leftarrow \mathsf{Dec}_{\{\!\{1\}\!\}}(\mathsf{ct}_\tau)$; • $\_, K'_0 \leftarrow \mathsf{GStack}_0.\mathsf{Pop}(\{\!\{1_L\}\!\})$; • $\_, K'_1 \leftarrow \mathsf{GStack}_1.\mathsf{Pop}(\{\!\{1_R\}\!\})$; • $\{\!\{1'_L\}\!\} := \{\!\{1\}\!\} \oplus K'_0 \oplus \mathsf{TrL}'_{0,\tau}$; • $\{\!\{1'_R\}\!\} := \{\!\{1\}\!\} \oplus K'_1 \oplus \mathsf{TrL}'_{1,\tau}$; 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 $\mathsf{TrL}'_{0,\tau} := \mathsf{Lbl}(K'_{0\ \tau}) \oplus \mathsf{Lbl}(\{\!\{1\}\!\}_\tau)$ and $\mathsf{TrL}'_{1,\tau} := \mathsf{Lbl}(K'_{1\ \tau}) \oplus \mathsf{Lbl}(\{\!\{1\}\!\}_\tau)$.
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 $\{\!\{\mathsf{st}_0, \mathsf{st}_1\}\!\}$.	

Fig. 3: GSwitch algorithm.

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 3 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 Finalize call (see Appendix D.2 of the online full version [30]).

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 $\mathbf{ctrlL} \xleftarrow{\$} \{0,1\}^\lambda$. 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 $\mathbf{IL}_b = \text{rand}()||\mathbf{ctrlL} \in \{0,1\}^{2t_{\max}\cdot\lambda}$; let $\mathbf{OL}_b = \text{rand}()||\mathbf{FinL} \in \{0,1\}^{t_{\max}\cdot w + m\cdot\lambda}$.
// 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 3, 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 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.

The cost of GSwitch is dominated by the garbled stacks which take $O(\log B)$ overhead. Thus, the construction in Figure 3 costs $O(\mathsf{B} \cdot (\lambda \cdot w_1 + w_2) \cdot \log \mathsf{B})$ bits, where $w_1 = |\mathsf{leaf}| + |\mathsf{addr}|$ and $w_2 = |L|$.

Security Proofs We defer the security proofs for GSwitch to Appendix D.3 of the online full version [30].

Leaf Switches (GLeafSwitch) We need a special (but simpler) type of switches for the leaf level. We defer the detailed description of the leaf switches to Appendix D.4 of the online full version [30].

5 Non-Recursive Garbled Memory (NRGRAM)

5.1 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 $\mathbf{OutL}$ 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.

5.2 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.

Additional building blocks. To construct our NRGRAM, we need a few additional building blocks, namely, garbled buckets denoted GBkt, garbled level rebuilder GRebuild, and garbled stash GStash. Their functionalities are roughly summarized below.

– GStash supports functions Read, Add, and Finalize, and it is parameterized by the maximum number of operations $\mathsf{GStash}.m$ and the word size $\mathsf{GStash}.w$.
– GBkt is parameterized by the number of entries m, the maximum number of operations $t_{\max}$, and the bit-width of each entry w. It supports $\mathsf{Init}(\{\!\{\mathbf{DB}\}\!\})$ which initializes the bucket with the list $\mathbf{DB}$, $\mathsf{Read}(\{\!\{\mathsf{addr}\}\!\})$ which looks up the entry addr, and Finalize. Similar to GSwitch, we write the input labels as $(\mathbf{InitL}, \mathbf{ReqL}, \mathbf{CtrlL})$ for $(\mathsf{Init}, \mathsf{Read}, \mathsf{Finalize})$ correspondingly, and the output labels of Read and Finalize are written as $(\mathbf{RdL}, \mathbf{FinL})$.

– $\mathsf{GRebuild}$ is parameterized by the time t and a corresponding level $\ell \in [0, \ell_{\max}]$ that depends on t. It takes in the stash the and the levels from 0 to ℓ, and then it outputs a new stash and new levels from 0 to $\min(\ell+1, \ell_{\max})$.

We defer the description of these building blocks to Appendix D.7, D.6, and C.1 of the online full version [30].

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 $\mathsf{GSwitch}^{V,t}$ (or $\mathsf{GLeafSwitch}^{V,t}$) 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:

– $\mathsf{GSwitch}^{V,t}$ 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 GStash^t.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 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 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 $\mathsf{GSwitch}^{V,t}$.Init but will call $\mathsf{GSwitch}^{V,t+1}$.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, $\mathsf{GRebuild}^t.\mathbf{OutL} := \{\mathsf{GBkt}^{(i,j),t+1}.\mathbf{InitL}\}_{i\in[0:\ell],j\in[2^i]}$.

5.3 Construction

We describe our NRGRAM construction in Figure 4, where the relevant data structures and how to choose the encoding labels were explained earlier in Section 5.2. In the step marked ($\diamondsuit$), the same variables $\{\!\{\mathsf{leaf}\}\!\}, \{\!\{\mathsf{addr}\}\!\}, [\![L]\!]$ are overwritten by the outcome of the call to $\mathsf{GSwitch}^{V,t}$.Switch($\{\!\{\mathsf{leaf}\}\!\}$, $\{\!\{\mathsf{addr}\}\!\}$, $[\![L]\!]$); keep in mind that the output variables do not have the same labels as the input variables, although the notation is the same.

[7] The leaf switches take no $\{\!\{\mathsf{addr}\}\!\}$ nor Finalize, and hence the parent GSwitch outputs $\{\!\{\mathsf{addr}\}\!\}$ or Finalize only to the children buckets (Figuer 4).

Fig. 4: Non-Recursive Garbled RAM (NRGRAM) construction.

The garbled data structures adopt the following parameters (see the supplementary for the parameters of GStash, $\mathsf{GLeafSwitch}$, and GBkt):

- 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

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

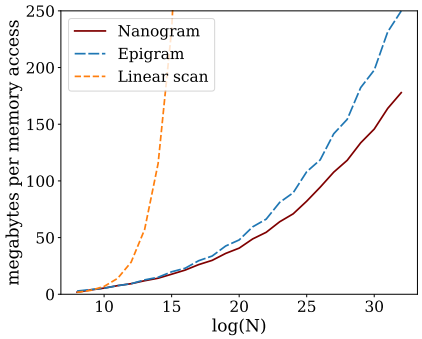
Proof of security. We defer the proof of security for our NRGRAM to Appendix E of the online full version [30].

6 Final Garbled RAM (GRAM) and Concrete Performance

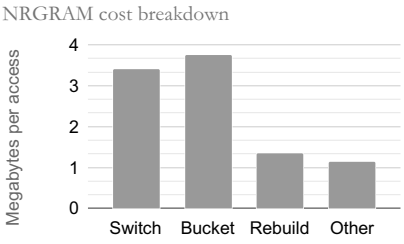
Full garbled RAM construction. Our final garbled RAM scheme is obtained by applying the standard recursion technique [33, 35] to the NRGRAM. The idea is to recursively store the position map in a smaller NRGRAM, and then store the position map of the position map in an even smaller NRGRAM, and do on. The recursion will stop in logarithmically many iterations as long as the block size is at least $C \log N$ for some appropriate constant C. We defer the detailed construction and proofs to Appendix F of the online full version [30].

Practical optimizations and concrete performance. In Appendix A.1 of the online full version [30], we propose several practical optimizations for our garbled RAM scheme.

We also developed a simulator to evaluate the concrete performance of our scheme. We defer the detailed explanation of the simulator and our experimental methology to Appendix A.2 of the online full version [30]. In Figure 5a, we compared the performance of NanoGRAM with that of the naïve linear-scan GRAM as well as EpiGRAM [24], where the word size $W = 128$ bits. Here, we use a standard platform-independent performance metric, i.e., *the garbled circuit size amortized to each memory access*, that is used in this line of work [24, 32]. Given



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



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

Fig. 5: Concrete performance of NanoGRAM. The y-axis is the size of the garbled RAM program amortized to each memory access.

the circuit size, we can estimate the runtime on typical computers using the results of earlier works [32]. 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 5b 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 optimizations — had we not performed these optimizations, 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

1. Applebaum, B.: Garbling xor gates "for free" in the standard model. In: TCC (2013). https://doi.org/10.1007/978-3-642-36594-2_10

2. Asharov, G., Komargodski, I., Lin, W.K., Nayak, K., Peserico, E., Shi, E.: OptORAMa: Optimal Oblivious RAM. In: Eurocrypt (2020). https://doi.org/10.1007/978-3-030-45724-2_14
3. Batcher, K.E.: Sorting networks and their applications. In: American Federation of Information Processing Societies: AFIPS Conference Proceedings (1968), https://doi.org/10.1145/1468075.1468121
4. Canetti, R., Chen, Y., Holmgren, J., Raykova, M.: Adaptive succinct garbled RAM or: How to delegate your database. In: TCC (2016). https://doi.org/10.1007/978-3-662-53644-5_3
5. Canetti, R., Holmgren, J.: Fully succinct garbled RAM. In: ITCS. pp. 169–178. ACM (2016). https://doi.org/10.1145/2840728.2840765
6. Chan, T.H., Nayak, K., Shi, E.: Perfectly secure oblivious parallel RAM. In: TCC (2018). https://doi.org/10.1007/978-3-030-03810-6_23
7. Chan, T.H., Shi, E.: Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In: TCC (2017). https://doi.org/10.1007/978-3-319-70503-3_3
8. Chan, T.H., Shi, E., Lin, W., Nayak, K.: Perfectly oblivious (parallel) RAM revisited, and improved constructions. In: ITC (2021). https://doi.org/10.4230/LIPIcs.ITC.2021.8
9. Chen, Y., Chow, S.S.M., Chung, K., Lai, R.W.F., Lin, W., Zhou, H.: Cryptography for parallel RAM from indistinguishability obfuscation. In: ITCS. pp. 179–190. ACM (2016). https://doi.org/10.1145/2840728.2840769
10. Choi, S.G., Katz, J., Kumaresan, R., Zhou, H.S.: On the security of the "free-xor" technique. In: TCC (2012). https://doi.org/10.1007/978-3-642-28914-9_3
11. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: TCC. pp. 144–163 (2011). https://doi.org/10.1007/978-3-642-19571-6_10
12. Fincher, D.: The curious case of benjamin button, film (2008)
13. Fletcher, C., Naveed, M., Ren, L., Shi, E., Stefanov, E.: Bucket ORAM: Single online roundtrip, constant bandwidth Oblivious RAM. Cryptology ePrint Archive, Report 2015/1065 (2015)
14. Garg, S., Lu, S., Ostrovsky, R.: Black-box garbled RAM. In: FOCS (2015). https://doi.org/10.1109/FOCS.2015.22
15. Garg, S., Lu, S., Ostrovsky, R., Scafuro, A.: Garbled ram from one-way functions. STOC (2015). https://doi.org/10.1145/2746539.2746593
16. Gentry, C., Goldman, K.A., Halevi, S., Jutla, C.S., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: PETS (2013). https://doi.org/10.1007/978-3-642-39077-7_1
17. Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., Wichs, D.: Garbled ram revisited. In: EUROCRYPT (2014). https://doi.org/10.1007/978-3-642-55220-5_23
18. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC (1987). https://doi.org/10.1145/28395.28416
19. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM (1996). https://doi.org/10.1145/233551.233553
20. Hazay, C., Lilintal, M.: Gradual gram and secure computation for ram programs. In: Security and Cryptography for Networks (2020). https://doi.org/10.1007/978-3-030-57990-6_12
21. Heath, D., Kolesnikov, V.: Stacked garbling - garbled circuit proportional to longest execution path. In: CRYPTO (2020). https://doi.org/10.1007/978-3-030-56880-1_27

22. Heath, D., Kolesnikov, V.: Logstack: Stacked garbling with O(b log b) computation. In: EUROCRYPT (2021). https://doi.org/10.1007/978-3-030-77883-5_1
23. Heath, D., Kolesnikov, V.: One hot garbling. In: CCS (2021). https://doi.org/10.1145/3460120.3484764
24. Heath, D., Kolesnikov, V., Ostrovsky, R.: Epigram: Practical garbled ram. In: EUROCRYPT (2022). https://doi.org/10.1007/978-3-031-06944-4_1
25. Kolesnikov, V., Mohassel, P., Rosulek, M.: Flexor: Flexible garbling for XOR gates that beats free-xor. In: CRYPTO (2014). https://doi.org/10.1007/978-3-662-44381-1_25
26. Kolesnikov, V., Schneider, T.: Improved Garbled Circuit: Free XOR Gates and Applications. In: ICALP (2008). https://doi.org/10.1007/978-3-540-70583-3_40
27. Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious ram lower bound! In: CRYPTO (2018). https://doi.org/10.1007/978-3-319-96881-0_18
28. Lu, S., Ostrovsky, R.: How to garble ram programs. In: EUROCRYPT (2013). https://doi.org/10.1007/978-3-642-38348-9_42
29. Lu, S., Ostrovsky, R.: Black-box parallel garbled RAM. In: CRYPTO (2017). https://doi.org/10.1007/978-3-319-63715-0_3
30. Park, A., Lin, W.K., Shi, E.: NanoGRAM: Garbled RAM with $\widetilde{O}(\log N)$ overhead. Cryptology ePrint Archive, Paper 2022/191 (2022), https://eprint.iacr.org/2022/191
31. Patel, S., Persiano, G., Raykova, M., Yeo, K.: Panorama: Oblivious ram with logarithmic overhead. In: FOCS (2018). https://doi.org/10.1109/FOCS.2018.00087
32. Rosulek, M., Roy, L.: Three halves make a whole? beating the half-gates lower bound for garbled circuits. In: CRYPTO (2021). https://doi.org/10.1007/978-3-030-84242-0_5
33. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: ASIACRYPT (2011). https://doi.org/10.1007/978-3-642-25385-0_11
34. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM – an extremely simple oblivious ram protocol. In: CCS (2013). https://doi.org/10.1145/2508859.2516660
35. Stefanov, E., Shi, E., Song, D.: Towards practical oblivious RAM. In: Network and Distributed System Security Symposium (NDSS) (2012)
36. Waksman, A.: A permutation network. J. ACM **15**(1), 159–163 (jan 1968). https://doi.org/10.1145/321439.321449
37. Wang, X.S., Chan, T.H.H., Shi, E.: Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In: CCS (2015). https://doi.org/10.1145/2810103.2813634
38. Wang, X.S., Nayak, K., Liu, C., Chan, T.H.H., Shi, E., Stefanov, E., Huang, Y.: Oblivious Data Structures. In: CCS (2014). https://doi.org/10.1145/2660267.2660314
39. Yao, A.C.C.: Protocols for secure computations (extended abstract). In: FOCS (1982). https://doi.org/10.5555/1382436.1382751
40. Yao, A.C.C.: How to generate and exchange secrets. In: FOCS (1986). https://doi.org/10.1109/SFCS.1986.25
41. Zahur, S., Evans, D.: Circuit Structures for Improving Efficiency of Security and Privacy Tools. In: IEEE S & P (2013). https://doi.org/10.1109/SP.2013.40
42. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole - reducing data transfer in garbled circuits using half gates. In: EUROCRYPT (2015). https://doi.org/10.1007/978-3-662-46803-6_8