Practical Garbled RAM
GRAM with $O(\log^2 n)$ Overhead

David Heath[1], Vladimir Kolesnikov[2], and Rafail Ostrovsky[3]

[1] heath.davidanthony@gatech.edu, Georgia Tech
[2] kolesnikov@gatech.edu, Georgia Tech
[3] rafail@cs.ucla.edu, UCLA

Abstract. Garbled RAM (GRAM) is a powerful technique introduced by Lu and Ostrovsky that equips Garbled Circuit (GC) with a sublinear cost RAM without adding rounds of interaction. While multiple GRAM constructions are known, none are suitable for practice, due to costs that have high constants and poor scaling.

We present the first GRAM suitable for practice. For computational security parameter κ and for a size-n RAM that stores blocks of size $w = \Omega(\log^2 n)$ bits, our GRAM incurs amortized $O(w \cdot \log^2 n \cdot \kappa)$ communication and computation per access. We evaluate the concrete cost of our GRAM; our approach outperforms trivial linear-scan-based RAM for as few as 512 128-bit elements.

Keywords: MPC, Garbled Circuits, Oblivious RAM, Garbled RAM

1 Introduction

Secure multiparty computation (MPC) allows mutually untrusting parties to compute functions of their combined inputs while revealing nothing but the outputs. MPC protocols traditionally consider functions encoded as circuits. While this does not limit expressivity, it does limit efficiency: many interesting computations are best expressed as RAM programs, not as circuits, and the reduction from RAM programs to circuits is expensive.

Fortunately, we can combine MPC with oblivious RAM (ORAM). ORAM is a technology that allows a client to outsource an encrypted database to a server; the client can then access the database while both (1) incurring only sublinear overhead and (2) hiding the access pattern from the server. By running an ORAM client inside MPC, we can augment circuits with random access memory. This powerful combination allows us to run RAM programs inside MPC.

Garbled Circuit (GC) is a foundational and powerful MPC technique that allows two parties to achieve secure computation while consuming only constant rounds of interaction. One party, the GC generator G, "encrypts" the circuit and sends it to the other party, the GC evaluator E. E is given an encryption of each party's input and steps through the circuit gate-by-gate under encryption. At each gate, E propagates encryptions of input wire values to encryptions of output wire values. Once E finishes, E and G can jointly decrypt the output wire values, revealing the circuit output.

It is natural to consider adding RAM to GC while preserving GC's constant rounds. However, the constant round requirement means that adding RAM to GC is seemingly more difficult than adding RAM to interactive protocols. Nevertheless, it is possible to run an ORAM client inside the GC and to let E play an ORAM server. This technique is called Garbled RAM (GRAM) [LO13].

While GRAM constructions are known [LO13,GHL$^+$14,GLOS15,GLO15], none are suitable for practice: existing constructions simply cost too much. All existing GRAMs suffer from at least two of the following problems:

- **Use of non-black-box cryptography.** [LO13] showed that GRAM can be achieved by evaluating a PRF *inside GC* in a non-black-box way. Unfortunately, this non-black-box cryptography is extremely expensive, and on each access the construction must evaluate the PRF *repeatedly.* [LO13] requires a circular-security assumption on GC and PRFs. Follow-up works removed this circularity by replacing the PRF with even more expensive non-black-box techniques [GHL$^+$14,GLOS15].
- **Factor-κ blowup.** Let κ denote the computational security parameter. In practical GC, we generally assume that we will incur factor κ overhead due to the need to represent each bit as a length-κ *encoding* (i.e. a GC label). However, existing GRAMs suffer from yet another factor κ. This overhead follows from the need to represent GC labels (which have length κ) *inside the GC* such that we can manipulate them with Boolean operations. The GC labels that encode a GC label together have length κ^2. In practice, where we generally use $\kappa = 128$, this overhead is intolerable.
- **High factor scaling.** Existing GRAMs operate as follows. First, they give an array construction that leaks access patterns to E. This leaky array already has high cost. Then, they compile this array access into GRAM using off-the-shelf ORAM. This compilation is problematic: off-the-shelf ORAMs require that, on each access, E access the leaky array a polylogarithmic (or more) number of times. Thus, existing GRAMs incur *multiplicative* overhead from the composition of the leaky array with the ORAM construction.

Prior GRAM works do not attempt to calculate their concrete or even asymptotic cost, other than to claim cost sublinear or polylogarithmic in n. In Supplementary Material A, we estimate the cost of prior GRAM. For a GRAM that stores 128-bit blocks, we conservatively estimate that the best prior GRAM breaks even with trivial linear-scan based GRAM when the RAM size reaches $\approx 2^{20}$ elements. Hence, our conservative estimate indicates that by the time it is worthwhile to use existing GRAM, each and every access requires a 4GB GC.

1.1 Contribution

We present the first practical garbled RAM. Our GRAM, which we call EpiGRAM, uses only $O(w \cdot \log^2 n \cdot \kappa)$ computation and communication per access. EpiGRAM circumvents all three of the above problems:

- **No use of non-black-box cryptography.** Our approach routes array elements using novel, yet simple, techniques. These techniques are light-weight, and non-black-box cryptography is not required.
- **No factor-κ blowup.** While we, like previous GRAMs, represent GC labels inside the GC itself, we give a novel generalization of existing GC gates that eliminates the additional factor κ overhead.
- **Low polylogarithmic scaling.** Like previous GRAMs, we present a leaky construction that reveals access patterns to E. However, we do not compile this into GRAM using off-the-shelf ORAM. Instead, we construct a custom ORAM designed with GC in mind. Our GRAM minimizes use of our leaky construction. The result is a highly efficient technique.

In the remainder of this paper we:

- Informally and formally describe the first practical GRAM. For an array with n elements each of size w such that $w = \Omega(\log^2 n)$, the construction incurs amortized $O(w \cdot \log^2 n \cdot \kappa)$ communication and computation per access.
- Prove our GRAM secure by incorporating it in a *garbling scheme* [BHR12]. Our scheme handles arbitrary computations consisting of AND gates, XOR gates, and array accesses. Our scheme is secure under a typical GC assumption: a circular correlation robust hash function [CKKZ12].
- Analyze EpiGRAM's concrete cost. Our analysis shows that EpiGRAM outperforms trivial linear-scan based RAM for as few as 512 128-bit elements.

2 Technical Overview

In this section, we explain our construction informally but with sufficient detail to understand our approach. This overview covers four topics:

- First, we explain a problem central to GRAM: *language translation*.
- Second, we informally explain our *lazy permutation network*, which is a construction that efficiently solves the language translation problem.
- Third, as a stepping stone to our full construction, we explain how to construct *leaky* arrays from the lazy permutation network. This informal construction securely implements an array with the caveat that we let E learn the array access pattern.
- Fourth, we upgrade the leaky array to full-fledged GRAM: the presented construction hides the access pattern from E.

2.1 The language translation problem

For each GC wire x_i the evaluator E holds one of two κ-bit strings: either X_i, which encodes a logical zero, or $X_i \oplus \Delta$, which encodes one. Meanwhile, G holds each such X_i and the global secret Δ. We refer to the wire-specific value X_i as the *language* of that wire, and to the pair $\langle X_i, X_i \oplus x_i \Delta \rangle$ jointly held by G and E as the *GC encoding*, or the *garbling*, of x_i. We present this notation formally in Section 4.4. To produce a garbled gate that takes as input a particular wire value x_i, G must know the corresponding language X_i. Normally this is not a problem: the structure of the circuit is decided statically, and G can easily track which languages go to which gates.

However, consider representing an array as a collection of such garbled labels. That is, there are n values x_i where E holds $X_i \oplus x_i\Delta$. Suppose that at runtime the GC requests access to a particular index α. We could use a static circuit to select x_α, but this would require an expensive linear-cost circuit. A different method is required to achieve the desired sublinear access costs.

Instead, suppose we disclose α to E in cleartext – we later add mechanisms that hide RAM indices from E. Since she knows α, E can jump directly to the αth wire and retrieve the value $X_\alpha \oplus x_\alpha\Delta$. Recall, to use a wire as input to a gate, G and E must agree on that wire's language. Unfortunately, it is *not possible* for G to predict the language X_α: α is computed during GC evaluation and, due to the constant round requirement, E cannot send messages to G.

Therefore, we instead allow G to select a fresh uniform language Y. If we can convey to E the value $Y \oplus x_\alpha\Delta$, then G will be able to garble gates that take the accessed RAM value as input, and we can successfully continue the computation.

Thus, our new goal is to *translate* the language X_α to the language Y. Mechanically, this translation involves giving to E the value $X_\alpha \oplus Y$. Given this, E simply XORs the translation value with her label and obtains $Y \oplus x_\alpha\Delta$. Keeping the circuit metaphor, providing such translation values to E allows her to take two wires – the wire out of the RAM and the wire into the next gate – and to solder these wires together *at runtime*. However, the problem of efficiently conveying these translation values remains.

In Supplementary Material B, we discuss natural attempts at solving the language translation problem. Translation can be achieved by a linear-sized gadget (suggesting dynamic conversion is possible), or by a non-black box PRF [LO13] (suggesting the ability to manipulate languages inside the GC). Our *lazy permutation network* (discussed next) achieves dynamic language translation more cheaply, but its underpinnings are the same: the network carefully manipulates languages inside the GC.

2.2 Lazy Permutations

Recall that our current goal is to translate GC languages. Suppose that the GC issues n accesses over its runtime. Further suppose that the GC accesses a *distinct location* on each access – in the end we reduce general RAM to a memory with this restriction. To handle the n accesses, we wish to convey to E n translation values $X_i \oplus Y_j$ where Y_j is G's selected language for the jth access.

What we need then is essentially a permutation on n elements that routes between RAM locations (with language X_i) and accesses (with language Y_j). However, a simple permutation network will not suffice, since at the time of RAM access j, the location of each subsequent access will, in general, not yet be known. Therefore, we need a *lazy* permutation whereby we can decide and apply the routing of the permutation one input at a time. We remind the reader

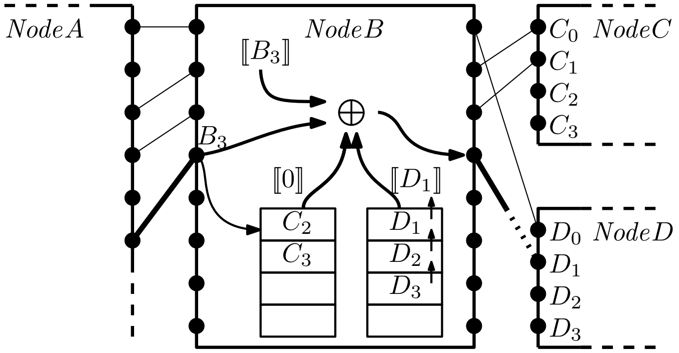

Fig. 1. An internal node of our lazy permutation network. We depict the fourth access to this node. The encoded input uses language B_3. We interpret the first encoded input bit as a flag that indicates to proceed left or right. Our objective is to forward the remaining input to either the left or right node. Each node stores two oblivious stacks that hold encodings of the unused languages of the two children. We conditionally pop both stacks. In this case, the left stack is unchanged whereas the right stack yields D_1, the next language for the target child. Due to the pop, the remaining elements in the right stack move up one slot. By XORing these values with an encoding of the input language, then opening the resulting value to E, we convert the message to the language of the target child, allowing E to solder a wire to the child.

that we assume that E knows each value α. I.e., we need only achieve a lazy permutation where E learns the permutation.

Given this problem, it may now be believable that algorithms and data structures exist such that the total cost is $O(\kappa \cdot n \cdot \text{polylog}(n))$, and hence only amortized $O(\kappa \cdot \text{polylog}(n))$ per access. Indeed we present such a construction. However, our solution requires that we apply this lazy permutation *to the GC languages themselves*, not to bits stored in the RAM. Thus, we need a logic in which we can encode GC languages: E must obliviously and authentically manipulate GC languages. GC gives us these properties, so we can encode languages bit-by-bit inside the GC. I.e., for a language of length w, we would add w GC wires, each of which would hold a single bit of the language.

Unfortunately, this bit-by-bit encoding of the languages leads to a highly objectionable factor κ blowup in the size of the GC: the encoding of a length-w language has length $w \cdot \kappa$. We later show that the factor κ blowup is unnecessary. Under particular conditions, existing GC gates can be generalized such that we can represent a length-w language using an encoding of only length w. These special and highly efficient GC gates suffice to build the gadgetry we need. We formalize the needed gate in Section 5.1.

The ability to encode languages inside the GC is powerful. Notice that since we can dynamically solder GC wires, and since wires can hold languages needed to solder other wires, we can arrange for E to repeatedly and dynamically lay down new wiring in nearly arbitrary ways.

With this high level intuition, we now informally describe our lazy permutation network. Let n be a power of two. Our objective is to route between the languages of n array accesses and the languages of n array elements.

G first lays out a full binary branching tree with n leaves. Each node in this tree is a GC with static structure. However, the inputs and outputs to these circuits are loose wires, ready to be soldered at runtime by E. At runtime, seeking to read array element x_α with language X_α into a wire with language Y, E begins at the root of the tree, which holds a GC encoding of the target language Y. (Note, G knows the target language Y of the j-th access, and can accordingly program the tree root.) Based on the GC encoding of the first bit of α, E is able to dynamically decrypt a translation value to either the left or the right child node. Now, E can solder wires to this child, allowing her to send to the child circuit both the encoding of Y and the remaining bits of α. E repeatedly applies this strategy until she reaches the αth leaf node. This leaf node is a special circuit that computes $\mathcal{C}(x) = x \oplus X_\alpha$ and then reveals the output to E.[4] Since we have pushed the encoding of Y all the way to this leaf, E obtains $Y \oplus X_\alpha$, the translation value that she needs to read x_α.

In yet more detail, each internal node on level k of the tree is a static circuit with $2^{\log n - k}$ loose sets of input wires. Each node maintains two *oblivious stacks* [ZE13]. The first stack stores encodings of the languages for the $2^{\log n - k - 1}$ loose input wires of the left child, and the second stack similarly stores languages for the right child (see Figure 1). On the j-th access and seeking to compute $Y_j \oplus X_\alpha$, E dynamically traverses the tree to leaf α (recall, we assume E knows α in cleartext), forwarding an encoding of Y_j all the way to the αth leaf. At each internal node, she uses a bit of the encoding of α to conditionally pop the two stacks, yielding an encoding of the language of the correct child. The static circuit uses this encoding to compute a translation value to the appropriate child.

By repeatedly routing inputs over the course of n accesses, we achieve a lazy permutation. Crucially, the routing between nodes is not decided until runtime.

This construction is affordable. Essentially the only cost comes from the oblivious stacks. For a stack that stores languages of length w, each pop costs only $O(w \cdot \log n)$ communication and computation (Section 5.2). Thus, the full lazy permutation costs only $O(w \cdot n \cdot \log^2 n)$ communication, which amortizes to sublinear cost per access. We describe our lazy permutation network in full formal detail in Section 5.3.

Our lazy permutation networks route the language of each RAM slot to the access where it is needed, albeit in a setting where E views the routing in cleartext. Crucially, the lazy permutation network avoids factor κ additional overhead that is common in GRAM approaches. To construct a secure GRAM, we build on this primitive and hide the RAM access pattern.

[4] Our actual leaf circuit is more detailed. See Sections 2.4 and 5.3.

2.3 Pattern-Leaking (Leaky) Arrays

As a stepping stone to full GRAM, we informally present an intermediate array which leaks access patterns. For brevity, we refer to it as *leaky* array. This construction handles arbitrary array accesses in a setting where E is allowed to learn the access pattern. We demonstrate a reduction from this problem to our lazy permutation network.

We never *formally* present the resulting construction. Rather, we explain the construction now for expository reasons: we decouple our explanation of *correctness* from our explanation of *obliviousness*. I.e., this section builds a correct GRAM that leaks the access pattern to E. The ideas for this leaky construction carry to our secure GRAM (Section 2.4).

Suppose the GC wishes to read index α. Recall that our lazy permutation network is a mechanism that can help translate GC languages: E can dynamically look up an encoding of the language X_α. However, because the network implements a *permutation*, it alone does not solve our problem: an array should allow multiple accesses to the same index, but the permutation can route each index to *only one* access. To complete the reduction, more machinery is needed.

To start, we simplify the problem: consider an array that handles at most n accesses. We describe an array that works in this restricted setting and later upgrade it to handle arbitrary numbers of accesses.

Logical indices → one-time indices. The key idea is to introduce a level of indirection. While the GC issues queries via logical indices α, our array stores its content according to a different indexing system: the content for each logical index α is stored at a particular *one-time index* p. As the name suggests, each one-time index may be written to and read at most once. This limitation ensures compatibility with a lazy permutation: since each one-time index is read only once, a permutation suffices to describe the read pattern.

Each one-time index can be read only once, yet each logical index can be read multiple times. Thus, over the course of n accesses, a given logical index might correspond to *multiple* one-time indices.

Neither party can *a priori* know the mapping between logical indices and one-time indices. However, to complete an access the GC must compute the relevant one-time index. Thus, we implement the mapping as a recursively instantiated *index map*.[5] The index map is itself a leaky array where each index α holds the corresponding one-time index p. We are careful that the index map is strictly smaller than the array itself, so the recursion terminates; when the next needed index map is small enough, we instantiate it via simple linear scans.

A leaky array with n elements each of size w and that handles at most n accesses is built from three pieces:

1. A block of $2n$ GC encodings each of size w called the one-time array. We index into the one-time array using one-time indices.

[5] Recursive index/position maps are typical in ORAM constructions, see e.g. [SvS$^+$13].

2. A size-$2n$ lazy permutation $\tilde{\pi}$ where each leaf i stores the language for one-time array slot i.
3. The recursively instantiated index map.

Let $\{\!\!\{x_i\}\!\!\}$ denote the GC encoding of bitstring x_i where G holds X_i and E holds $X_i \oplus x_i\Delta$ (see also Section 4.4). Suppose the parties start with a collection of n encodings $\{\!\!\{x_0\}\!\!\}, ..., \{\!\!\{x_{n-1}\}\!\!\}$ which they would like to use as the array content. The parties begin by sequentially storing each value $\{\!\!\{x_i\}\!\!\}$ in the corresponding one-time index i. The initial mapping from logical indices to one-time indices is thus statically decided: each logical index i maps to one-time index i. The parties recursively instantiate the index map with content $\{\!\!\{0\}\!\!\}, ..., \{\!\!\{n-1\}\!\!\}$.

When the GC performs its j-th access to logical index $\{\!\!\{\alpha\}\!\!\}$, we perform the following steps:

1. The parties recursively query the index map using input $\{\!\!\{\alpha\}\!\!\}$. The result is a one-time index $\{\!\!\{p\}\!\!\}$. The parties simultaneously write back into the index map $\{\!\!\{n+j\}\!\!\}$, indicating that α will next correspond to one-time index $n+j$.
2. The GC reveals p to E in cleartext. This allows E to use the lazy permutation network $\tilde{\pi}$ to find a translation value for the pth slot of the one-time array.
3. E jumps to the pth slot of the array and translates its language, soldering the value to the GC and completing the read. Note that the GC may need to access index α again, so the parties perform the next step:
4. The parties write back to the $(n+j)$-th slot of the one-time array. If the access is a read, they write back the just-read value. Otherwise, they write the written value.

In this way, the parties can efficiently handle n accesses to a leaky array.

Handling more than n accesses. If the parties need more than n accesses, a reset step is needed. Notice that after n accesses, we have written to each of the $2n$ one-time indices (n during initialization and one per access), but we have only read from n one-time indices. Further notice that on an access to index α, we write back a new one-time index for α; hence, it must be the case that the n remaining unread one-time array slots hold the current array content.

Going beyond n accesses is simple. First, we one-by-one read the n array values in the sequential *logical* order (i.e. with $\alpha = 0, 1, .., n-1$), flushing the array content into a block $\{\!\!\{x_0\}\!\!\}, ..., \{\!\!\{x_{n-1}\}\!\!\}$. Second, we initialize a new leaky array data structure, using the flushed block as its initial content. This new data structure can handle n more accesses. By repeating this process every n accesses, we can handle arbitrary numbers of accesses.

Summarizing the leaky array. Thus, we can construct an efficient garbled array, which leaks access patterns. Each access to the leaky array costs amortized $O(w \cdot \log^2 n \cdot \kappa)$ bits of communication, due to the lazy permutation network. We emphasize the key ideas that carry over to our secure GRAM:

- We store the array data according to *one-time indices*, not according to logical indices. This ensures compatibility with our lazy permutation network.
- We recursively instantiate an *index map* that stores the mapping from logical indices to one-time indices.
- We store the GC languages of the underlying data structure in a lazy permutation network such that E can dynamically access slots.
- Every n accesses, we *flush* the current array and instantiate a fresh one.

2.4 Garbled RAM

In Section 2.3 we demonstrated that we can reduce random access arrays to our lazy permutation network, so long as E is allowed to learn the access pattern. In this section we strengthen that construction by hiding the access patterns, therefore achieving secure GRAM.

Note that this strengthening is clearly possible, because we can simply employ off-the-shelf ORAM. In ORAM, the server learns a physical access pattern, but the ORAM protocol ensures that these physical accesses together convey no information about the logical access pattern. Thus, we can use our leaky array to implement physical ORAM storage, implement the ORAM client inside the GC, and the problem is solved.

We are not content with this solution. The problem is that our leaky array already consumes $O(\log^2 n)$ overhead, due to lazy permutations. In ORAM, each logical access is instantiated by at least a logarithmic number of physical reads/writes. Thus, compiling our leaky array with off-the-shelf ORAM incurs *at least* an additional $O(\log n)$ *multiplicative factor*. In short, this off-the-shelf composition is expensive.

We instead directly improve the leaky array construction (Section 2.3) and remove its leakage. This modification incurs only additive overhead, so our GRAM has the same asymptotic cost as the leaky array: $O(w \cdot \log^2 n \cdot \kappa)$ bits per access.

The key idea of our full GRAM is as follows: In regular ORAM, we assume that the client is significantly weaker than the server. In our case, too, the GC – which plays the client – *is* much weaker than E – who plays the server. However, we have a distinct advantage: the GC generator G can act as a powerful advisor to the GC, directly informing most of its decisions.

More concretely, our GRAM carefully arranges that the locations of almost all of the physical[6] reads and writes are decided *statically* and are independent of the logical access pattern. Thus, G can *a priori* track the static schedule and prepare for each of the static accesses. Our GRAM incurs $O(\log^2 n)$ physical reads/writes per logical access. However, only a *constant number*[7] of these reads cannot be predicted by G, as we will soon show.

[6] I.e. reads and writes to the lowest level underlying data structure, where access patterns are visible to E.

[7] To be pedantic, if we account for recursively instantiated index maps, each map incurs this constant number of unpredictable reads, so there are total a logarithmic number of unpredictable reads.

Each physical read/write requires that G and E agree on the GC language of the accessed element. For each statically decided read/write, this agreement is reached trivially. Therefore, we only need our lazy permutation network for reads that G cannot predict. There are only a constant number of these, so we only need a constant number of calls to the lazy permutation network.

Upgrading the leaky array. We now informally describe our GRAM. Our description is made by comparison to the leaky array described in Section 2.3.

In the leaky array, we stored all $2n$ one-time indices in a single block. In our GRAM, we instead store the $2n$ one-time indices across $O(\log n)$ *levels* of exponentially increasing size: each level i holds 2^{i+1} elements, though some levels are vacant. As we will describe later, data items are written to the smallest level and then slowly move from small levels to large levels. Each populated level of the GRAM holds 2^i one-time-indexed data items and 2^i *dummies*. Dummies are merely encodings of zero. Each level of the GRAM is stored shuffled. The order of items on each level is unknown to E but, crucially, *is known to* G. This means that at all times G knows which one-time index is stored where and knows which elements are dummies.

In the leaky array, E was pointed directly to the appropriate one-time index. In our GRAM, we need to hide the identity of the level that holds the appropriate index. Otherwise, since elements slowly move to larger levels, E will learn an approximation of the time at which the accessed element was written. Hence we arrange that E will read from *each* level on each access. However, all except one of these accesses will be to a dummy, and the indices of the accessed dummies are *statically scheduled by* G. More precisely, G *a priori* chooses one dummy on *each* populated level and enters their addresses as input to the GC. The GC then conditionally replaces one dummy address by the real address, then reveals each address to E. (Note that G *does not know* which dummy goes unaccessed – we discuss this later.)

In the leaky array and when accessing logical index α, we used the index map to find corresponding one-time index p. p was then revealed to E. In our GRAM, it is not secure for E to learn one-time indices corresponding to accesses. Thus, we introduce a new uniform permutation π of size $2n$ that is held by G and secret from E. Our index map now maps each index $\{\!\!\{\alpha\}\!\!\}$ to the corresponding *permuted* one-time index $\{\!\!\{\pi(p)\}\!\!\}$. We can safely reveal $\pi(p)$ to E – the sequence of such revelations is indistinguishable from a uniform permutation.

In the leaky array, we used the lazy permutation network $\tilde{\pi}$ to map each one-time index p to a corresponding GC language. Here, we need two changes:

1. Instead of routing p to the metadata corresponding to p, we instead route $\pi(p)$ to the metadata corresponding to p. G can arrange for this by simply initializing the content of the lazy permutation in permuted order.
2. We slowly move one-time indexed array elements from small levels to large levels (we have not yet presented how this works). Thus, each one-time index no longer corresponds to a single GC language. Instead, each one-time index now corresponds to a *collection* of physical addresses. Moreover, each time

we move a one-time index to a new physical address, it is crucial to security that we encode the data with a different GC language. Fortunately, we ensure that G knows the entire history of each one-time index. Thus, he can garble a circuit that takes as input the number of accesses so far and outputs the *current* physical address and GC language. We place these per-one-time-index circuits at the leaves of a lazy permutation network.

Remark 1 (Indices). Our GRAM features three kinds of indices:

- *Logical indices* α refer to simple array indices. The purpose of the GRAM is to map logical indices to values.
- Each time we access a logical index, we write back a corresponding value to a fresh *one-time index* p. Thus, each logical index may correspond to multiple one-time indices. The mapping from logical indices to one-time indices is implemented by the recursively instantiated *index map*.
- One-time indices are not stored sequentially, but rather are stored permuted such that we hide access patterns from E. A *physical address* @ refers to the place where a one-time index p is currently held. Because we repeatedly move and permute one-time indices, each one-time index corresponds to multiple physical addresses. The mapping from one-time indices to physical addresses is known to G and is stored in a lazy permutation network.

In the leaky array and on access j, we write back an element to one-time index $n + j$. In our GRAM, we similarly perform this write. We initially store this one-time index in the smallest level. Additionally, the parties store a fresh dummy in the smallest level. After each write, the parties permute a subset of the levels of RAM using a traditional permutation network. The schedule of permutations – see next – is carefully chosen such that the access pattern is hidden but cost is low. Over the course of n accesses, the n permutations together consume only $O(n \cdot \log^2 n)$ overhead.

The permutation schedule. Recall that we arrange the RAM content into $O(\log n)$ levels of exponentially increasing size. After each access, G applies a permutation to a subset of these levels. These permutations prevent E from learning the access pattern.

Recall that on each access, E is instructed to read from each populated level. All except one of these reads is to a dummy. Further recall that after being accessed once, a one-time index is never used again. Thus, it is important that each dummy is similarly accessed at most once. Otherwise, E will notice that doubly-accessed addresses must hold dummies.

Since we store only 2^i dummies on level i, level i can only support 2^i accesses: after 2^i accesses it is plausible that all dummies have been exhausted. To continue processing, G therefore re-permutes the level, mixing the dummies and real elements such that the dummies can be safely reused. More precisely, on access j we collect those levels i such that 2^i divides j. Let k denote the largest such i. We concatenate each level $i \leq k$ together into a block of size 2^{k+1} and permute its contents into level $k + 1$ (this level is guaranteed to be vacant). This leaves each level $i \leq k$ vacant and ready for new data to flow up. Now that the data has been permuted, it is safe to once again use the shuffled dummies, since they are shuffled and each is given a new GC language.

As a security argument, consider E's view of a particular level i over all 2^i accesses between permutations. Each such access could be to a dummy or to a real element, but these elements are uniformly shuffled. Hence, E's view can be simulated by uniformly sampling, without replacement, a sequence of 2^i indices.

Remark 2 (Permutations). Our RAM features three kinds of permutations:

- $\tilde{\pi}$ is a *lazy* permutation whose routing is revealed to E over the course of n accesses. The lazy permutation allows E to efficiently look up the physical address and language for the target one-time index.
- π is a uniform permutation chosen by G whose sole purpose is to ensure that $\tilde{\pi}$ does not leak one-time indices to E. Let π' denote the *actual* routing from RAM accesses to one-time indices. E does not learn π', but rather learns $\tilde{\pi} = \pi' \circ \pi$. Since π is uniform, $\tilde{\pi}$ is also uniform.
- $\pi_0, ..., \pi_{n-1}$ is a sequence of permutations chosen by G and applied to levels of GRAM. These ensure that the physical access pattern leaks nothing to E.

Accounting for the last dummy per access. One small detail remains. Recall that on each access, G statically chooses a dummy on each of the $O(\log n)$ levels. E will be pointed to each of these dummies, save one: E will not read the dummy on the same level as the real element. The identity of the real element is dynamically chosen, so G cannot know which dummy is not read. The parties must somehow account for the GC language of the unread dummy to allow E to proceed with evaluation. (We expand on this need in a moment.)

This accounting is easily handled by a simple circuit $\mathcal{C}_{hide}$. $\mathcal{C}_{hide}$ takes as input an encoding of the real physical address and outputs an encoding of the language of the unaccessed dummy.

We now provide more detail (which can be skipped at the first reading) explaining why E must recover an encoding of the language of the unaccessed dummy. Suppose the real element is on level j. G selects $O(\log n)$ dummy languages D_i for this access, and E reads one label in each language $D_{i \neq j}$, and reads the real value. To proceed, G and E must obtain the real value in some agreed language, and this language must depend on all languages D_i (since G cannot know which dummy was not read). Therefore, D_j must be obtained and used by E as well. In even more detail, in the mind of G, the "output" language includes the languages D_i XORed together; to match this, in addition to XORing all labels she already obtained, E XORs in the encoding of the missing dummy language. The validity of this step relies heavily on Free XOR [KS08].

The high level procedure. To conclude our overview, we enumerate the steps of the RAM. Consider an arbitrary access to logical index α.

1. E first looks up α's current one-time index p by consulting the index map. The index map returns an encoding of $\pi(p)$ where π is a uniform permutation that hides one-time indices from E.
2. The GC reveals $\pi(p)$ to E in cleartext such that she can route the lazy permutation $\tilde{\pi}$. E uses $\tilde{\pi}$ to route the current RAM time to a leaf circuit that computes encodings of the appropriate physical address @ and GC language. Let ℓ denote the RAM level that holds address @.
3. A per-access circuit $\mathcal{C}_{hide}$ is used to compute (1) encodings of physical addresses of dummies on each populated level $i \neq \ell$ and (2) the GC language of the dummy that *would* have been accessed on level ℓ, had the real element been on some other level.
4. The GC reveals addresses to E and E reads each address. E XORs the results together. (Recall, dummies are garblings of zero.) Each read value is a GC label with a distinct language. To continue, G and E must agree on the language of the resulting GC label. G can trivially account for the GC language of each dummy except for the unaccessed dummy. E XORs on the encoded language for the accessed element and the encoded language for the unaccessed dummy. This allows E to solder the RAM output to the GC such that computation can continue.
5. Parties write back an encoding either of the just-accessed-element (for a read) or of the written element (for a write). This element is written to the smallest level. Parties also write a fresh dummy to the smallest level.
6. G applies a permutation to appropriate RAM levels.
7. After the nth access, E flushes the RAM by reading each index without writing anything back, then initializes a new RAM with the flushed values.

We formalize our GRAM in Section 5.4.

3 Related Work

Garbled RAM. [LO13] were the first to achieve sublinear random access in GC. As already mentioned, their GRAM evaluates a PRF inside the GC and also requires a circular-security assumption.

This circularity opened the door to further improvements. [GHL$^+$14] gave two constructions, one that assumes identity-based-encryption and a second that assumes only one-way functions, but that incurs super-polylogarithmic overhead. [GLOS15] improved on this by constructing a GRAM that simultaneously assumes only one-way functions and that achieves polylog overhead. Both of these works avoid the [LO13] circularity assumption, but are expensive because they repeatedly evaluate cryptographic primitives inside the GC.

[GLO15] were the first to achieve a GRAM that makes only black-box use of crypto-primitives. Our lazy permutation network is inspired by [GLO15]: the authors describe a network of GCs, each of which can pass the program control flow to one of several other circuits. In this way they translate between GC languages. Our approach improves over the [GLO15] approach in several ways:

- The [GLO15] GRAM incurs factor κ blowup when passing messages through their network of GCs. Our lazy permutation network avoids this blowup.
- [GLO15] uses a costly probabilistic argument. Each node of their network is connected to a number of other nodes; this number scales with the statistical security parameter. The authors show that the necessary routing can be achieved at runtime with overwhelming probability.[8] This approach uses a network that is *significantly* larger than is needed for any particular routing, and most nodes are ultimately wasted. In contrast, our lazy permutation network is direct. Each node connects to exactly two other nodes, and all connections are fully utilized over n accesses.
- [GLO15] compile their GRAM using off-the-shelf ORAM, incurring multiplicative overhead between their network of GCs and the ORAM. We build a custom RAM that makes minimal use of our lazy permutation network.

In this work, we focus on RAM access in the standard GC setting. A number of other works have explored other dimensions of GRAM, such as parallel RAM access, adaptivity, and succinctness [CCHR16,CH16,LO17,GOS18].

Practical GC and ORAM. Due to space, we defer discussion of works in the areas of practical GC and ORAM to Supplementary Material C.

4 Preliminaries, Notation, and Assumptions

4.1 Common Notation

- G is the circuit generator. We refer to G as he/him.
- E is the circuit evaluator. We refer to E as she/her.
- We denote by $\langle x, y \rangle$ a pair of values where G holds x and E holds y.
- κ is the computational security parameter (e.g. 128).
- We write $x \triangleq y$ to denote that x is defined to be y.
- $\overset{c}{=}$ is the computational indistinguishability relation.
- $x \leftarrow y$ denotes that variable x is assigned to value y; x can later be reassigned.
- We generally use n to denote the number of elements and w to denote the bit-width of those elements.
- $[x]$ denotes the natural numbers $0, ..., x-1$.

Our construction is a garbling scheme [BHR12], not a protocol. I.e., our construction is merely a tuple of procedures that can be plugged into GC protocols. However, it is often easier to think of G and E as participating in a semi-honest protocol. Thus, we often write that the parties “send messages”. We make two notes about this phrasing:

- We will never write that E sends a message to G: all information flows from G to E. In this way, we preserve the constant round nature of GC.
- ‘G sends x to E’ formally means that (1) our garbling procedure appends x to the GC and (2) our evaluation procedure extracts x from the GC.

[8] The [GLO15] probabilistic argument *requires* that indices be accessed randomly. I.e., the [GLO15] leaky array cannot be used except by plugging it into ORAM.

4.2 Cryptographic Assumptions

We use the Free XOR technique [KS08], so we assume a circular correlation robust hash function H [CKKZ12,ZRE15]. In practice, we instantiate H using fixed-key AES [GKWY20].

4.3 Garbling Schemes

A *garbling scheme* [BHR12] is a method for securely computing a class of circuits in constant rounds. A garbling scheme is *not* a protocol; rather, it is a tuple of procedures that can be plugged into a variety of protocols.

Definition 1 (Garbling Scheme). *A* garbling scheme *for a class of circuits* $\mathbb{C}$ *is a tuple of procedures:*

$$(\mathit{Gb}, \mathit{En}, \mathit{Ev}, \mathit{De})$$

where (1) Gb maps a circuit $\mathcal{C} \in \mathbb{C}$ *to a garbled circuit* $\tilde{\mathcal{C}}$*, an input encoding string e, and an output decoding string d; (2) En maps an input encoding string e and a cleartext bitstring x to an encoded input; (3) Ev maps a circuit* $\mathcal{C}$*, a garbled circuit* $\tilde{\mathcal{C}}$*, and an encoded input to an encoded output; and (4) De maps an output decoding string d and encoded output to a cleartext output string.*

A garbling scheme must be *correct* and may satisfy any combination of *obliviousness*, *privacy*, and *authenticity* [BHR12]. We include formal definitions of these properties in Supplementary Material F. Our scheme satisfies each definition and hence can be plugged into GC protocols.

4.4 Garblings and Sharings

We work with two kinds of encodings of logical values: ‘garblings’ and simple XOR shares. Garblings correspond to the traditional notion of garbled labels; i.e., a garbling is a length-κ value held by each party.

Recall from Section 2 that we manipulate languages inside the GC. This is why we work also with simple XOR sharings: we use XOR sharings to encode and move languages inside the GC. We define notation for both types of shares, and we emphasize the compatibility of garblings and sharings.

Garblings are Free XOR-style garbled circuit labels [KS08]. G samples a uniform value $\Delta \in \{0,1\}^{\kappa-1}1$. I.e., Δ is uniform except that the least significant bit is one. Δ is *global* to the entire computation. A garbling of $x \in \{0,1\}$ is a tuple $\langle X, X \oplus x\Delta \rangle$, where the first element (here, X) is held by G, and the second by E.

Definition 2 (Garbling). *Let* $x \in \{0,1\}$ *be a bit. Let* $X \in \{0,1\}^\kappa$ *be a bitstring held by* G*. We say that the pair* $\langle X, X \oplus x\Delta \rangle$ *is a* garbling *of* x *over (usually implicit)* $\Delta \in \{0,1\}^{\kappa-1}1$*. We denote a garbling of* x *by writing* $\{\!|x|\!\}$*:*

$$\{\!|x|\!\} \triangleq \langle X, X \oplus x\Delta \rangle$$

Definition 3 (Sharing). *Let $x, X \in \{0,1\}$ be two bits. We say that the pair $\langle X, X \oplus x\rangle$ is a* sharing *of x. We denote a sharing of x by writing $[\![x]\!]$:*

$$[\![x]\!] \triangleq \langle X, X \oplus x\rangle$$

We refer to G's share X as the *language* of the garbling (resp. sharing). Except in specific circumstances, we use uniformly random languages both for garblings and for sharings.

Note, XOR is homomorphic over garblings [KS08] and sharings:

$$\{\!\!\{a\}\!\!\} \oplus \{\!\!\{b\}\!\!\} = \{\!\!\{a \oplus b\}\!\!\} \qquad [\![a]\!] \oplus [\![b]\!] = [\![a \oplus b]\!]$$

We extend our garbling and sharing notation to vectors of values. That is, a garbling (resp. sharing) of a vector is a vector of garblings (resp. sharings):

$$\{\!\!\{a_0, ..., a_{n-1}\}\!\!\} \triangleq (\{\!\!\{a_0\}\!\!\}, ..., \{\!\!\{a_{n-1}\}\!\!\}) \qquad [\![a_0, ..., a_{n-1}]\!] \triangleq ([\![a_0]\!], ..., [\![a_{n-1}]\!])$$

Remark 3 (Length of garblings/sharings). Garblings are longer than sharings. I.e., let $x \in \{0,1\}$ be a bit. Then $\{\!\!\{x\}\!\!\}$ is a pair of length-κ strings held by G and E. Meanwhile, $[\![x]\!]$ is a pair of bits held by G and E.

Remark 4 (Sharings contain garblings). Notice that the space of sharings contains the space of garblings. Indeed, this will be important later: we will in certain instances reinterpret a garbling $\{\!\!\{x\}\!\!\}$ as a sharing $[\![x\Delta]\!]$. This will allow us to operate on the garbling as if it is a sharing.

We frequently deal with values that are known to a particular party. We write x^G (resp. x^E) to denote that x is a value known to G (resp. to E) in cleartext. E.g., $\{\!\!\{x^E\}\!\!\}$ indicates a garbling of x where E knows x.

Operations on Sharings/Garblings.

- $\{\!\!\{x\}\!\!\} \mapsto [\![x]\!]$. Recall that G ensures that the least significant bit of Δ is one. Suppose each party takes the least significant bit of his/her part of $\{\!\!\{x\}\!\!\}$:

$$\begin{aligned} \mathit{lsb}(\{\!\!\{x\}\!\!\}) = \mathit{lsb}(\langle X, X \oplus x\Delta\rangle) &\triangleq \langle \mathit{lsb}(X), \mathit{lsb}(X \oplus x\Delta)\rangle \\ = \langle \mathit{lsb}(X), \mathit{lsb}(X) \oplus x \cdot \mathit{lsb}(\Delta)\rangle &= \langle \mathit{lsb}(X), \mathit{lsb}(X) \oplus x\rangle = [\![x]\!] \end{aligned}$$

  That is, if both parties compute *lsb* on their parts of a garbling, the result is a valid sharing of the garbled value. This idea was first used to implement the classic point and permute technique.
- $[\![x]\!] \mapsto x^E$ and $\{\!\!\{x\}\!\!\} \mapsto x^E$. G can open the cleartext value of a sharing by sending his share to E. Similarly, we can open a garbling by first computing *lsb* (see above) and then opening the resulting share.
- $x^G \mapsto [\![x]\!]$ and $x^G \mapsto \{\!\!\{x\}\!\!\}$. G can easily introduce fresh inputs. Specifically, let x be a bit chosen by G and unknown to E. The parties can construct $\langle x, 0\rangle = [\![x]\!]$. Similarly, the parties can construct $\langle x\Delta, 0\rangle = \{\!\!\{x\}\!\!\}$.
- $\{\!\!\{x\}\!\!\} \cdot \{\!\!\{y\}\!\!\} \mapsto \{\!\!\{x \cdot y\}\!\!\}$. Garblings support AND gates. This operation can be implemented using two ciphertexts [ZRE15] (or 1.5 ciphertexts [RR21]).

- Input:
  - G inputs a permutation on n elements π.
  - A garbled array $\llbrace x_0, ..., x_{n-1} \rrbrace$ where $x_i \in \{0,1\}^w$.
- Output:
  - The permuted array $\llbrace \pi(x_0, ..., x_{n-1}) \rrbrace$.

Fig. 2. Interface to the procedure *G-permute* which permutes n values using a permutation π chosen by G. For power of two n, permuting n garbled values each of length w costs $w \cdot (n \log n - n + 1) \cdot \kappa$ bits of communication via a permutation network [Wak68].

- $x^G \cdot \llbrace y \rrbrace \mapsto \llbrace x \cdot y \rrbrace$. It is possible to instantiate a cheaper AND gate if G knows in cleartext one of the arguments. This operation can be implemented using one ciphertext [ZRE15].
- $\llbrace x^E \rrbrace \cdot \llbracket y \rrbracket \mapsto \llbracket x \cdot y \rrbracket$. This novel operation scales a vector of sharings by a garbling whose cleartext value is known to E. Section 5.1 gives the procedure.
- $\llbrace x \rrbrace \cdot y^G \mapsto \llbracket x \cdot y \rrbracket$. This operation follows simply from the above scaling procedure. See Section 5.1.

4.5 Oblivious Permutation

We permute garbled arrays using permutations chosen by G. A permutation on $n = 2^k$ width-w elements can be implemented using $w(n \log n - n + 1)$ AND gates via a classic construction [Wak68]. Since G chooses the permutation, we can use single ciphertext AND gates and implement the permutation for only $w \cdot (n \log n - n + 1) \cdot \kappa$ bits. Figure 2 lists the interface to this procedure.

5 Approach

In this section we formalize the approach described in Section 2. Our formalism covers four topics:

- Section 5.1 formalizes our generalized GC gates. These gates allow us to avoid the factor-κ blowup that is common to prior GRAMs.
- Section 5.2 uses these new gates to modify an existing pop-only stack construction [ZE13]. Our modified pop-only stacks leak their access pattern to E but can efficiently store GC languages.
- Section 5.3 uses pop-only stacks to formalize our lazy permutation network.
- Section 5.4 builds on the lazy permutation network to formalize our GRAM.

We package the algorithms and definitions in this section into a garbling scheme [BHR12] that we call EpiGRAM. EpiGRAM handles arbitrary circuits with AND gates, XOR gates, and array accesses, and is defined as follows:

Construction 1 (EpiGRAM)**.** EpiGRAM *is a garbling scheme (Definition 1) that handles circuits with four kinds of gates:*

- *XOR gates take as input two bits and output the XOR of the two inputs.*

- *AND gates take as input two bits and output the AND of the two inputs.*
- *ARRAY gates are parameterized over power of two n and positive integer w. The gate outputs a zero-initialized array of n elements each of width w.*
- *ACCESS gates take as input (1) an array A, (2) a $(\log n)$-bit index α, (3) a w-bit value y to store in the case of a write, and (4) a bit r that indicates if this is a read or write. The gate outputs $A[\alpha]$. As a side effect, A is mutated:*

$$A[\alpha] \leftarrow \begin{cases} y & \textit{if } r = 0 \\ A[\alpha] & \textit{otherwise} \end{cases}$$

The garbling scheme procedures are defined as follows:

- *En and De are standard; formally, our scheme is* projective *[BHR12], which allows us to implement En and De as simple maps between cleartext and encoded values. We formalize En and De in Supplementary Material E.*
- *Ev and Gb each proceed gate-by-gate through the circuit. For each XOR gate, each procedure XORs the inputs [KS08]. For each AND gate, the procedures compute the half-gates approach [ZRE15]. For each ARRAY gate, Gb (resp. Ev) invokes G's (resp. E's) part of the array initialization procedure (Figure 9). For each ACCESS gate, Gb (resp. Ev) invokes G's (resp. E's) part of the array access procedure (Figure 10).*

In Supplementary Material F and G, we prove lemmas and theorems that together imply the following result:

Theorem 1 (Main Theorem). *If H is a circular correlation robust hash function, then* EpiGRAM *is a* correct, oblivious, private, *and* authentic *garbling scheme. For each ACCESS gate applied to an array of n elements each of size $w = \Omega(\log^2 n)$, Gb outputs a GC of amortized size $O(w \cdot \log^2 n \cdot \kappa)$ and both Gb and Ev consume amortized $O(w \cdot \log^2 n \cdot \kappa)$ computation.*

5.1 Avoiding Factor κ Blowup

Recall from Section 2 that we avoid the factor-κ overhead that is typical in GRAMs. We now give the crucial operation that enables this improvement.

Our operation scales a vector of κ sharings by a garbled bit whose value is known to E. The scaled vector remains hidden from E. The operation computes $\{\!\!\{x^E\}\!\!\} \cdot [\![y]\!] \mapsto [\![x \cdot y]\!]$ for $y \in \{0,1\}^\kappa$ (see Figure 3). Crucially, the operation only requires that G send to E κ total bits. While this presentation is novel, the procedure in Figure 3 is a simple generalization of techniques given in [ZRE15]. This generalization allows us to scale an encoded GC language of length w (when $w = c \cdot \kappa$ for some c) for only w bits. This is how we avoid factor-κ blowup.

Formally, we have a *vector space* where the vectors are sharings and the scalars are garblings whose value is known to E. Vector space operations cannot compute arbitrary functions of sharings, but they can arbitrarily move sharings around. These data movements suffice to build our lazy permutation network.

Given Figure 3, we can also compute $\{\!\!\{x\}\!\!\} \cdot y^G \mapsto [\![x \cdot y]\!]$ for $y \in \{0,1\}^\kappa$:

- Input:
  - A garbled bit known to E: $\{\!|x^E|\!\}$.
  - A shared vector $[\![y]\!]$ for $y \in \{0,1\}^\kappa$.
- Output:
  - A sharing of the scaled vector $[\![x \cdot y]\!]$.
- Procedure $\{\!|x^E|\!\} \cdot [\![y]\!]$:
  - Parties agree on a gate-specific nonce i.
  - Let $\langle X, X \oplus x\Delta \rangle = \{\!|x^E|\!\}$.
  - Let $\langle Y, Y \oplus y \rangle = [\![y]\!]$.
  - G computes and sends to E $row \triangleq H(X \oplus \Delta, i) \oplus H(X, i) \oplus Y$.
  - E computes the following:

$$\begin{aligned}
&H(X \oplus x\Delta, i) \oplus x \cdot (row \oplus (Y \oplus y)) \\
&= H(X \oplus x\Delta, i) \oplus \begin{cases} row \oplus (Y \oplus y) & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} H(X \oplus \Delta, i) \oplus (H(X \oplus \Delta, i) \oplus H(X, i) \oplus Y) \oplus Y \oplus y & \text{if } x = 1 \\ H(X, i) & \text{otherwise} \end{cases} \\
&= H(X, i) \oplus x \cdot y
\end{aligned}$$

  - Parties output (respective shares of) $\langle H(X, i), H(X, i) \oplus x \cdot y \rangle = [\![x \cdot y]\!]$.

Fig. 3. Scaling a shared κ-bit vector by a garbling where E knows in cleartext the scalar. Scaling a κ-bit sharing requires that G send to E κ bits. We prove the construction secure when G's share of the vector $[\![y]\!]$ is either (1) a uniform bitstring Y or (2) a bitstring $z\Delta$ for $z \in \{0,1\}$. The latter case arises when G introduces a garbled input.

- Procedure $\{\!|x|\!\} \cdot y^G$:
  - Parties compute $[\![x]\!] = lsb(\{\!|x|\!\})$. Let $\langle X, X \oplus x \rangle = [\![x]\!]$.
  - G introduces inputs $\{\!|X|\!\}$, $[\![y]\!]$ and $[\![X \cdot y]\!]$.
  - Parties compute $\{\!|X|\!\} \oplus \{\!|x|\!\} = \{\!|X \oplus x|\!\}$. Note that E knows $X \oplus x$.
  - Parties compute (using Figure 3) and output:

$$\{\!|X \oplus x|\!\} \cdot [\![y]\!] \oplus [\![X \cdot y]\!] = [\![(X \oplus x) \cdot y]\!] \oplus [\![X \cdot y]\!] = [\![x \cdot y]\!]$$

This procedure is useful in our lazy permutation network and in the $\mathcal{C}_{hide}$ circuit.

5.2 Pop-only Oblivious Stacks

Our lazy permutation network uses pop-only oblivious stacks [ZE13], a data structure with a single pop operation controlled by a garbled bit. If the bit is one, then the stack indeed pops. Otherwise, the stack returns an encoded zero and is left unchanged. Typically, both the data stored in the stack *and* the access pattern are hidden. For our purposes, we only need a stack where the stored data is hidden from E, but where E learns the access pattern.

[ZE13] gave an efficient circuit-based stack construction that incurs only $O(\log n)$ overhead per pop. This construction stores the data across $O(\log n)$

- Input:
  - A block of n elements $\llbracket x_0, ..., x_{n-1} \rrbracket$ where $x_i \in \{0,1\}^w$.
- Output:
  - A capacity n stack $\mathit{Stack}(x_0, ..., x_{n-1})$.

- Input:
  - A size n stack $\mathit{Stack}(x_0, ..., x_{n-1})$.
  - A garbled bit known to E $\{\!|p^E|\!\}$ that indicates whether or not to pop.
- Output:
  - The popped value $\llbracket p \cdot x_0 \rrbracket$.
  - The updated stack:

$$\begin{cases} \mathit{Stack}(x_1, ..., x_{n-1}, 0^w) & \text{if } p = 1 \\ \mathit{Stack}(x_0, ..., x_{n-1}) & \text{otherwise} \end{cases}$$

Fig. 4. Interface to stack procedures *stack-init* (top) and *pop* (bottom). For a stack of size n with width-w entries, parties locally initialize using $O(w \cdot n)$ computation; each pop costs amortized $O(w \cdot \log n)$ communication and computation.

levels of exponentially increasing size; larger levels are touched exponentially less often than smaller levels, yielding low logarithmic overhead.

If E is allowed to learn the access pattern, we can implement the [ZE13] construction where the stack holds arbitrary sharings, not just garblings. This is done by replacing AND gates – which move data towards the top of the stack – with our scaling gate (Figure 3). Since we simply replace AND gates by scaling gates, we do not further specify. A modified stack with n elements each of width w costs amortized $O(w \cdot \log n)$ bits of communication per pop.

Construction 2 (Pop-only Stack)**.** *Let $x_0, ..., x_{n-1}$ be a n elements such that $x_i \in \{0,1\}^w$. $\mathit{Stack}(x_0, ..., x_{n-1})$ is a* pop-only stack *of elements $x_0, ..., x_{n-1}$. Pop-only stacks support the procedures stack-init and pop (Figure 4).*

5.3 Lazy Permutations

Recall from Section 2 that our lazy permutation network allows E to look up an encoded physical address and an encoded language for the needed RAM slot. The network is a binary tree where each inner node holds two pop-only oblivious stacks. Each inner node forwards messages to its children. Once a message is forwarded all the way to a leaf, the leaf node interprets the message as (1) an encoding of the current RAM time and (2) an encoding of an output language. This leaf node accordingly computes encodings of the appropriate physical address and language, then translates these to the output language. The encoded address and language are later used to allow E to read from RAM.

Inner nodes. For simplicity of notation, let level 0 denote the tree level that holds the leaves; level $\log n$ holds the root. Consider an arbitrary inner node i on

- Input:
  - Let i denote the node id and k denote the tree level. Level 0 holds leaves; level $\log n$ holds the root.
  - Parties input two stacks $s_0 = \mathit{Stack}(L^{\ell}_{2i}, ...)$ and $s_1 = \mathit{Stack}(L^{r}_{2i+1}, ...)$ such that each language L^b_a is an independent uniform string unknown to E.
  - Parties input message $\llbracket m \rrbracket$ such that $m \in \{0,1\}^{k \cdot \kappa + w}$
  - E inputs a bit d indicating if m should be sent to the left or right child.
- Output:
  - E outputs $L^{(\bar{d}\ell + dr)}_{2i+d} \oplus m'$ for $m' \in \{0,1\}^{(k-1)\kappa + w}$. I.e., she outputs a share that encodes the last $(k-1)\kappa + w$ bits of the incoming message m and that is encoded by a language for child d.
  - Parties output updated stacks $\mathit{Stack}(L^{\ell+\bar{d}}_{2i}, ...)$ and $\mathit{Stack}(L^{r+d}_{2i+1}, ...)$.
- Procedure $\mathit{inner}(s_0, s_1, \llbracket m \rrbracket, d)$:
  - Parties parse $\llbracket m \rrbracket$ as $\{\!|d^E|\!\}, \llbracket m' \rrbracket$.
  - Parties pop both stacks (Figure 4):

$$(\llbracket \bar{d} \cdot L^{\ell}_{2i} \rrbracket, s'_0) = \mathit{pop}(s_0, \{\!|\bar{d}|\!\}) \qquad (\llbracket d \cdot L^{r}_{2i+1} \rrbracket, s'_1) = \mathit{pop}(s_1, \{\!|d|\!\})$$

  - Parties compute:

$$\llbracket \bar{d} \cdot L^{\ell}_{2i} \oplus d \cdot L^{r}_{2i+1} \oplus m' \rrbracket = \llbracket L^{(\bar{d}\ell + dr)}_{2i+d} \oplus m' \rrbracket$$

  - G opens his share to E and E outputs $L^{(\bar{d}\ell + dr)}_{2i+d} \oplus m'$.
  - Parties output s'_0 and s'_1.

Fig. 5. Procedure for inner nodes of a lazy permutation network.

level k. This node can 2^k times receive a message $\llbracket m \rrbracket$ of a fixed, arbitrary length. On each message, the node strips the first κ bits from the message and interprets them as the garbling of a bit $\{\!|d|\!\}$. d is a direction indicator: if $d = 0$, then the node forwards the remaining message to its left child; otherwise it forwards to its right child. Over its lifetime, the inner node forwards 2^{k-1} messages to its left child and 2^{k-1} messages to its right child. Crucially, the *order* in which a node distributes its 2^k messages to its children is not decided until runtime.

Each of the 2^k messages are sharings with a particular language. I.e., the jth message $\llbracket m_j \rrbracket$ has form $\langle L_j, L_j \oplus m_j \rangle$ where each language L_j is distinct. The node must convert each message to a language next expected by the target child.

Assume that a particular node has so far forwarded ℓ messages to its left child and r messages to its right child. Let L^b_a denote the bth input language for node a. Note that the current language is thus $L^{\ell+r}_i$ and the language expected by the left (resp. right) child is L^{ℓ}_{2i} (resp. L^{r}_{2i+1}).

To forward m_j based on d, the node computes the following translation value:

$$\left\llbracket \bar{d} \cdot L^{\ell}_{2i} \oplus d \cdot L^{r}_{2i+1} \right\rrbracket = \llbracket L^{(\bar{d}\ell + dr)}_{2i+d} \rrbracket \qquad (1)$$

To compute the above, node i maintains two oblivious pop-only stacks (see Section 5.2) of size 2^{k-1}. The first stack stores, in order, sharings of the 2^{k-1} languages for the left child. The second stack similarly stores languages for the right

- Input:
    - Let this node be leaf $\pi(p)$ where π is a permutation chosen by G.
    - G inputs the storage metadata (Definition 4) $\mathcal{M}_p$ for one-time index p.
    - Parties input $\{\!\!\{T\}\!\!\}$, a garbling of the current RAM time.
    - Parties input $[\![Y]\!]$, a sharing of an output language such that Y is uniform.
- Output:
    - Let $(t_i^p, @_i^p, X_i^p)_{i\in[\log n]} = \mathcal{M}_p$. Let t_j^p be the largest metadata timer such that $t_j^p \leq T$. E outputs $Y \oplus (@_j^p \cdot \varDelta, X_j^p)$. I.e., she outputs a sharing of the appropriate physical address and language for one-time index p.
- Procedure *leaf* $(\mathcal{M}_p, \{\!\!\{T\}\!\!\}, [\![Y]\!])$:
    - Parties set $\{\!\!\{@\}\!\!\} \leftarrow \{\!\!\{@_0^p\}\!\!\}$ and $[\![X]\!] \leftarrow [\![X_0^p]\!]$.
    - For each $i \in \{1..\log n - 1\}$ parties compute $\{\!\!\{t_i^p \leq T\}\!\!\}$ via a Boolean circuit.
    - For each $i \in \{1..\log n - 1\}$ the parties update $[\![X]\!]$:

$$\begin{aligned}
[\![X]\!] &\leftarrow [\![X]\!] \oplus \{\!\!\{t_i^p \leq T\}\!\!\} \cdot (X_{i-1}^p \oplus X_i^p) \\
&= [\![X \oplus (t_i^p \leq T) \cdot (X_{i-1}^p \oplus X_i^p)]\!] && G \text{ knows } (X_{i-1}^p \oplus X_i^p) \\
&= \begin{cases} [\![X \oplus X \oplus X_i^p]\!] & \text{if } t_i^p \leq T \\ [\![X]\!] & \text{otherwise} \end{cases} && (t_i^p \leq T) \Rightarrow (t_{i-1}^p \leq T) \text{ (Defn. 4)} \\
&= \begin{cases} [\![X_i^p]\!] & \text{if } t_i^p \leq T \\ [\![X]\!] & \text{otherwise} \end{cases}
\end{aligned}$$

      We elaborate the above step carefully to show this conditional update can be achieved using efficient sharing procedures given in Section 5.1.

    - For each $i \in \{1..\log n - 1\}$ the parties update $\{\!\!\{@\}\!\!\}$ via a Boolean circuit:

$$\{\!\!\{@\}\!\!\} \leftarrow \begin{cases} \{\!\!\{@_i^p\}\!\!\} & \text{if } t_i^p \leq T \\ \{\!\!\{@\}\!\!\} & \text{otherwise} \end{cases}$$

    - Let $[\![m]\!] \triangleq \{\!\!\{@\}\!\!\}, [\![X]\!]$ be the concatenated output. Then parties compute $[\![m \oplus Y]\!]$ and G opens his share to E.
    - E outputs $m \oplus Y = Y \oplus (@_j^p \cdot \varDelta, X_j^p)$.

Fig. 6. Procedure for leaf nodes of a lazy permutation network.

child. By popping both stacks based on $\{\!\!\{d\}\!\!\}$, the node computes Equation (1). Figure 5 specifies the formal procedure for inner nodes.

Leaf nodes. Once a message has propagated from the root node to a leaf, we are ready to complete a lookup. Each leaf node of the lazy permutation network is a static circuit that outputs the encoding of a physical address and a language.

As the parties access RAM, G repeatedly permutes the physical storage to hide the access pattern from E. Each one-time index p has $O(\log n)$ different physical addresses and languages; the needed address and language depends on how many accesses have occurred. Thus, each leaf node must conditionally output one of $O(\log n)$ values depending on how many accesses have occurred.

- Input:
  - G inputs a uniform size-n permutation π.
  - G inputs storage metadata $\mathcal{M}_p$ (Definition 4) for each one-time index p.
- Output:
  - Parties output a size-n lazy permutation $\tilde{\pi}$.
- Procedure $\tilde{\pi}$-*init*$(\pi, \mathcal{M}_{p\in[n]})$:
  - G and E consider a full binary tree with n leaves.
  - For each node i on tree level k, G uniformly samples 2^k languages $L_i^{j\in[2^k]}$.
  - For each inner node i on level k of the tree, G and E initialize two stacks:

    $$s_i^\ell \triangleq \mathit{stack\text{-}init}\left(L_{2i}^{j\in[2^{k-1}]}\right) \qquad s_i^r \triangleq \mathit{stack\text{-}init}\left(L_{2i+1}^{j\in[2^{k-1}]}\right)$$

  - For each inner node i on level k of the tree and for each $j \in [2^k]$ G runs the inner node (Figure 5):

    $$(\cdot, s_i^\ell, s_i^r) \leftarrow \mathit{inner}(s_i^\ell, s_i^r, L_i^j, \cdot)$$

    E does not run these procedures. Instead, she receives and stores the 2^k GCs.
  - For each leaf node i, parses L_i^0 into strings L_T, L_Y of appropriate length. G runs the leaf (Figure 6):

    $$\mathit{leaf}(\mathcal{M}_{\pi^{-1}(i)}, L_T, L_Y)$$

    E does not run this procedure. Instead, she receives and stores the GC.
  - The parties output $\tilde{\pi} \triangleq (\llbracket L_0^{j\in[n]} \rrbracket, (s^\ell_{i\in[n-1]}, s^r_{i\in[n-1]}))$

Fig. 7. Lazy permutation network initialization. When initializing with leaves that store languages of length w, G sends to E a GC of size $O(w \cdot n \cdot \log^2 n)$ bits.

G chooses all permutations and storage languages before the first RAM access. Hence, G can precompute metadata indicating which one-time index will be stored where and with what language at which point in time:

Definition 4 (Storage Metadata). *Consider a one-time index p. The* storage metadata $\mathcal{M}_p$ *for one-time index p is a sequence of* $\log n$ *three-tuples:*

$$\mathcal{M}_p \triangleq (t_i^p, @_i^p, L_i^p)_{[i\in\log n]}$$

where each t_i^p is a natural number that indicates a point in time, $@_i^p$ is a physical address, and L_i^p is a uniform language. Each time $t_i \leq t_{i+1}$.

In our construction, each one-time index p may have fewer than $\log n$ corresponding physical addresses. G pads storage metadata by repeating the last entry until all $\log n$ slots are filled. G uses the storage metadata for each one-time index to configure each leaf. Figure 6 specifies the procedure for leaf nodes.

Putting the network together. We now formalize the top level lazy permutation network. To instantiate a new network, G and E agree on a size n and

- Input:
  - A size n lazy permutation network $\tilde{\pi}$.
  - A garbled index $\{\!\!|\pi(p)|\!\!\}$ such that $\pi(p)$ has not yet been routed.
  - The current RAM time T.
- Output:
  - A physical address $\{\!\!|@^p|\!\!\}$.
  - A shared language $\llbracket X^p \rrbracket$.
  - The updated lazy permutation network (i.e., where $\pi(p)$ has been routed).
- Procedure *route*$(\tilde{\pi}, \{\!\!|\pi(p)|\!\!\}, T)$:
  - Let v denote the number of times $\tilde{\pi}$ has already been used.
  - G and E parse the input lazy permutation network:

    $$\left(\llbracket L_0^{j\in[n]} \rrbracket, (s^\ell_{i\in[n-1]}, s^r_{i\in[n-1]})\right) = \tilde{\pi}$$

  - G samples a uniform value Y with length appropriate for the output; the parties trivially hold $\llbracket Y \rrbracket$. The parties also hold $\llbracket L_0^v \rrbracket$.
  - Parties collect $\llbracket m \rrbracket \triangleq \{\!\!|\pi(p)|\!\!\}, \{\!\!|T|\!\!\}, \llbracket Y \rrbracket$ and then compute $\llbracket L_0^v \oplus m \rrbracket$; G opens his share to E such that E holds $L_0^v \oplus m$.
  - Recall from Figure 7 that at initialization, E stored 2^k GCs for each level k node. Let E initialize $M \leftarrow L_0^v \oplus m$. E now traverses the tree from root to leaf $\pi(p)$. At each node i on the path to $\pi(p)$, G invokes:

    $$(M, s^\ell_j, s^r_j) \leftarrow \mathit{inner}(s^\ell_j, s^r_j, M, d)$$

    where j is the id of the ith node on the path to $\pi(p)$ and d is the ith bit of $\pi(p)$. To perform each invocation, E loads in the jth GC stored at initialization. This propagates E's share of $\{\!\!|T|\!\!\}$ and $\llbracket Y \rrbracket$ to leaf $\pi(p)$.
  - E invokes (using the appropriate GC) the leaf node procedure:

    $$Y \oplus (@^p \cdot \Delta, X^p) \leftarrow \mathit{leaf}(\cdot, \{\!\!|T|\!\!\}, \llbracket Y \rrbracket)$$

  - The parties output the updated $\tilde{\pi}$.
  - The parties compute and output:

    $$\langle Y, Y \oplus (@^p \cdot \Delta, X^p) \rangle = \llbracket @^p \cdot \Delta, X^p \rrbracket = \{\!\!|@^p|\!\!\}, \llbracket X^p \rrbracket$$

Fig. 8. Procedure to route one value through a lazy permutation network.

a width w and G provides storage metadata, conveying the information that should be stored at the leaves of the network. From here, G proceeds node-by-node through the binary tree, fully garbling each node. E receives all such GCs from G, but crucially she does not yet begin to evaluate. Instead, she stores the GCs for later use, remembering which GCs belong to each individual node.

Recall that G selects a uniform permutation π that prevents E from viewing the one-time index access pattern: when the GC requests access to one-time index p, E is shown $\pi(p)$. Now, let us consider the ith access to the network. At the time of this access, a garbled index $\{\!\!|\pi(p)|\!\!\}$ is given as input by the parties.

G selects a uniform language Y to use as the output language, and the parties trivially construct the sharing $\llbracket Y \rrbracket$. The parties then concatenate the message

$[\![m_i]\!] \triangleq \{\!\!\{\pi(p)\}\!\!\}, \{\!\!\{T\}\!\!\}, [\![Y]\!]$ where T is the number of RAM writes performed so far. Let L_0^i denote the ith input language for the root node 0. The parties compute $[\![L_0^i]\!] \oplus [\![m_i]\!]$ and G sends his resulting share, giving to E a valid share of m_i with language configured for the root node. E now feeds this value into the the tree, starting from the root node and traversing the path to leaf $\pi(p)$. Note that G does not perform this traversal, since he already garbled all circuits.

Each inner node strips off one garbled bit of $\pi(p)$. This propagates the message to leaf $\pi(p)$. Finally, the leaf node computes the appropriate physical address and language for one-time index p and translates them to language Y. Let $Y \oplus (@^p \cdot \Delta, L^p)$ denote E's output from the leaf node. The parties output:

$$\langle Y, Y \oplus (@^p \cdot \Delta, L^p)\rangle = [\![@^p \cdot \Delta, L^p]\!] = \{\!\!\{@^p\}\!\!\}, [\![L^p]\!]$$

Thus, the parties successfully read an address and a language from the network.

Construction 3 (Lazy Permutation Network)**.** *Let n be a power of two. A size-n* lazy permutation network *$\tilde{\pi}$ is a two-tuple consisting of:*

1. *Sharings of the input languages to the root node $[\![L_0^{j\in[n]}]\!]$.*
2. *$2n-2$ stacks belonging to the $n-1$ inner nodes, $s^\ell_{i\in[n-1]}$ and $s^r_{i\in[n-1]}$.*

Here, each input language $L_0^{j\in[n]}$ and each language stored in each stack is an independently sampled uniform string. Lazy permutation networks support initialization (Figure 7) and routing of a single input (Figure 8).

5.4 Our GRAM

We formalize our GRAM on top of our lazy permutation network:

Construction 4 (GRAM)**.** *Let n – the RAM size – be a power of two and let w – the word size – be a positive integer. Let $x_0, ..., x_{n-1}$ be n values such that $x_i \in \{0,1\}^w$. Then Array$(x_{i\in[n]})$ denotes a size-n GRAM holding the content $x_{i\in[n]}$. Concretely, a GRAM is a tuple consisting of:*

1. *A timer T denoting the number of writes performed so far.*
2. *A sequence of languages $\mathcal{X}$ held by G and used as the languages for the permuted RAM content. Each language has length $w \cdot \kappa$, sufficient to encode a single garbled word.*
3. *A size-$2n$ uniform permutation π held by G.*
4. *A sequence of $n+1$ uniform permutations $\pi_0, ..., \pi_n$ held by G and used to permute the physical storage. These hide the RAM access pattern from E.*
5. *A size-$2n$ lazy permutation $\tilde{\pi}$.*
6. *A recursively instantiated RAM called the* index map *that maps each logical index α to $\pi(p)$: the (permuted) one-time index where α is currently saved. For each recursive RAM of size n, we instantiate the index map with word size $w = 2(\log n + 1)$. To bound the recursion, we use a linear-scan based RAM when instantiating a index map that stores only $O(w \cdot \log^2 n)$ bits.*

- Input:
  - Let n denote a number of elements and let w denote the width of each element. The parties input a vector $\{\!\{x_0, ..., x_{n-1}\}\!\}$ where $x_i \in \{0,1\}^w$
- Output:
  - A length n random access array $Array(x_{i\in[n]})$.
- Procedure *array-init*($\{\!\{x_{i\in[n]}\}\!\}$):
  - Parties initialize the timer T to n, indicating the n initial writes.
  - G schedules all accesses and computes his needed metadata:

    $$(\mathcal{X}, \mathcal{M}_{p\in[2n]}, \pi_{i\in[n+1]}) \leftarrow G\text{-}schedule(n, w)$$

  - G uniformly samples a size-$2n$ permutation π.
  - Parties instantiate the lazy permutation network: $\tilde{\pi} \leftarrow \tilde{\pi}\text{-}init(\pi, \mathcal{M}_{p\in[2n]})$
  - G and E recursively initialize the index map with content $\{\!\{0, 1, ..., n-1\}\!\}$, indicating that each index i starts in one-time index i:

    $$index\text{-}map \leftarrow array\text{-}init(\{\!\{0, 1, ..., n-1\}\!\})$$

  - Parties zero initialize the stash and each of the $\log n + 2$ levels of storage.
  - Parties store the initial data $\{\!\{x_{i\in[n]}\}\!\}$ on level $\log n - 1$.[a]

[a] This is a simple trick. On each access, we shuffle RAM levels (see Figures 10 and 14). By initializing the content on level $\log n - 1$, we ensure that the first access will shuffle the n items with n dummies and place them on level $\log n$.

Fig. 9. RAM initialize.

7. $\log n + 2$ levels *of physical storage where level i is a garbling of size $w \cdot 2^{i+1}$. Each level i is either vacant or stores 2^i real elements and 2^i dummies. The physical storage is permuted according to permutations $\pi_0, ..., \pi_n$.*
8. *A garbling of size $2w$ called the* stash. *Parties write back to the stash; on each access, items are immediately moved from the stash into a level of storage.*

GRAMs support initialization (Figure 9) and access (Figure 10).

Our top level garbling scheme is defined with respect to this data structure; EpiGRAM makes explicit calls to *array-init* (Figure 9) and *access* (Figure 10).

We call attention to *G-schedule*, *shuffle*, *flush*, and *hide*:

- *G-schedule* is a local procedure run by G where he plans ahead for the next n accesses. Specifically, G selects uniform permutations on storage, chooses uniform languages with which to store the RAM content, and computes the storage metadata $\mathcal{M}_p$ for each one-time index $p \in [2n]$. Supplementary Material D gives the explicit interface to *G-schedule*.
- *shuffle* describes how G permutes levels of storage. By doing so, we ensure that the revealed physical addresses give no information to E. *shuffle* is a straightforward formalization of the permutation schedule given in Section 2.4 and is formalized in Supplementary Material D.
- After each n-th access, we invoke *flush* (Figure 11) to reinitialize GRAM. We also mention that our proof of correctness (see Supplementary Material

- Input:
  - A length n array $A = \mathit{Array}(x_0, ..., x_{n-1})$.
  - A garbled index $\{\!\!\{\alpha\}\!\!\}$ such that $\alpha \in \{0,1\}^{\log n}$.
  - A garbled value $\{\!\!\{y\}\!\!\}$ to store in the case of a write.
  - A garbled bit $\{\!\!\{r\}\!\!\}$ that indicates if this is a read; else this is a write.
- Output:
  - The indexed value $\{\!\!\{x_\alpha\}\!\!\}$.
  - The updated array $\mathit{Array}(x_0, ..., x_{\alpha-1}, (r \cdot x_\alpha \oplus \bar{r} \cdot y), x_{\alpha+1}, ..., x_{n-1})$.
- Procedure $\mathit{access}(A, \{\!\!\{\alpha\}\!\!\}, \{\!\!\{y\}\!\!\}, \{\!\!\{r\}\!\!\})$:
  - Parties appropriately permute levels of storage: $A \leftarrow \mathit{shuffle}(A)$
  - If $T = 2n$ then the parties reinitialize and try again, returning that result:

    $$\mathit{access}(\mathit{array\text{-}init}(\mathit{flush}(A)), \{\!\!\{\alpha\}\!\!\}, \{\!\!\{y\}\!\!\}, \{\!\!\{r\}\!\!\})$$

    Otherwise, the parties continue as follows:
  - Parties recursively access the index map and update the one-time index for index α by writing back a garbling $\{\!\!\{\pi(T)\}\!\!\}$ (G knows $\pi(T)$):

    $$\{\!\!\{\pi(p)\}\!\!\} \leftarrow \mathit{access}(\mathit{index\text{-}map}, \{\!\!\{\alpha\}\!\!\}, \{\!\!\{\pi(T)\}\!\!\}, \{\!\!\{0\}\!\!\})$$

  - G opens his share of $\{\!\!\{\pi(p)\}\!\!\}$ to reveal $\pi(p)$ to E.
  - E uses $\tilde{\pi}$ to route time T to leaf $\pi(p)$. This returns the current physical address and language corresponding to p.

    $$(\{\!\!\{@\}\!\!\}, [\![X]\!]) \leftarrow \mathit{route}(\tilde{\pi}, \{\!\!\{\pi(p)\}\!\!\}, T)$$

  - For each populated storage level i, G chooses a previously unaccessed dummy element with address $@'_i$ and language D_i.
  - Let j denote the level that holds @. Parties compute (Figure 15):

    $$(\{\!\!\{@_i\}\!\!\}, [\![D_j]\!]) \leftarrow \mathit{hide}(@'_i, D_i, \{\!\!\{@\}\!\!\})$$

    I.e., *hide* computes one physical address per populated storage level.
  - G reveals to E each physical address $@_i$ by sending his share.
  - E reads each physical address and XORs the values together. I.e., E reads each dummy language $D_{i \neq j}$ and the desired element $X \oplus x_\alpha \Delta$. This yields:

    $$\left(\bigoplus_{i \neq j} D_i\right) \oplus X \oplus x_\alpha \Delta$$

  - Let $\langle L, L \oplus X\rangle = [\![X]\!]$ and $\langle L', L' \oplus D_j\rangle = [\![D_j]\!]$. Parties compute and output:

    $$\begin{aligned} &\left\langle L \oplus L' \oplus \left(\bigoplus_i D_i\right), L \oplus X \oplus L' \oplus D_j \oplus \left(\bigoplus_{i \neq j} D_i\right) \oplus X \oplus x_\alpha \Delta \right\rangle \\ = &\left\langle L \oplus L' \oplus \left(\bigoplus_i D_i\right), L \oplus L' \oplus \left(\bigoplus_i D_i\right) \oplus x_\alpha \Delta \right\rangle = \{\!\!\{x_\alpha\}\!\!\} \end{aligned}$$

  - Parties compute $\{\!\!\{r \cdot x_\alpha \oplus \bar{r} \cdot y\}\!\!\}$ and place their shares in the first slot of the stash. Parties place $\{\!\!\{0\}\!\!\}$, a fresh dummy, in the second slot of the stash.
  - Parties increment the timer: $T \leftarrow T + 1$.

Fig. 10. RAM access.

- Input:
  - A length n array $A = \mathit{Array}(x_0, ..., x_{n-1})$.
- Output:
  - The flushed content $\{\!\!\{ x_0, ..., x_{n-1} \}\!\!\}$.
- Procedure *flush*(A):
  - Parties recursively flush the index map:

    $$\{\!\!\{ \pi(p_0), ..., \pi(p_{n-1}) \}\!\!\} \leftarrow \mathit{flush}(\mathit{index\text{-}map})$$

  - For each $i \in [n]$ the parties route time T to leaf $\pi(p_i)$, returning the current physical address and language corresponding to p_i:

    $$(\{\!\!\{ @_i \}\!\!\}, \llbracket X_i \rrbracket) \leftarrow \mathit{route}(\tilde{\pi}, \{\!\!\{ \pi(p_i) \}\!\!\}, T)$$

    When flushing, each level $i \neq \log n + 1$ is vacant, so we need not use extra machinery to hide the accessed level: E knows each item is on level $\log n + 1$.
  - G reveals to E each physical address $@_i$ by sending his share.
  - E reads each address $@_i$, yielding $X_i \oplus x_i \Delta$.
  - For each i, let $\langle L_i, L_i \oplus X_i \rangle = \llbracket X_i \rrbracket$. Parties compute and output:

    $$\langle L_i, L_i \oplus X_i \oplus X_i \oplus x_i \Delta \rangle = \{\!\!\{ x_i \}\!\!\}$$

Fig. 11. *flush* is a helper procedure used to reset the array after n accesses. *flush* recovers the n array elements and places them into a contiguous block.

F) defines correctness of the GRAM data structure with respect to *flush*: a GRAM is *valid* if we can flush and recover its content.

- On each access, *hide* picks a dummy on each storage level, the conveys to E (1) a physical address on each level of storage and (2) a sharing of the language of the unaccessed dummy. The precise procedure is formalized in Supplementary Material D.

With these four helper procedures defined, we formalize GRAM initialization (Figure 9) and GRAM access (Figure 10). Initialization is straightforward, and GRAM access is a formalization of the high level procedure given in Section 2.4.

6 Evaluation

In this section, we analyze EpiGRAM's performance. We leave implementation and low-level optimization as important future work.

To estimate cost, we implemented a program that modularly computes the communication cost of each of EpiGRAM's subcomponents. E.g., a permutation network on n width- $w$elements uses $w \cdot (n \log n - n + 1)$ ciphertexts [Wak68].

Figure 12 fixes the word size w to 128. That is, each RAM slot stores 128 *garbled* bits. We plot the estimated communication cost as a function of n. For comparison, we also plot the cost of a linear scan; a linear scan on n elements of width w and while using [ZRE15] AND gates can be achieved for (slightly more

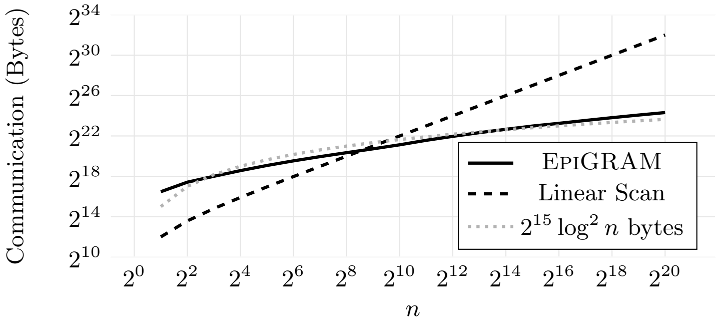

Fig. 12. Estimated concrete communication cost of our GRAM. We fix the word size $w = 128$ and plot per-access amortized communication as a function of n.

than) $2 \cdot w \cdot (n - 1)$ ciphertexts. We also plot the function $2^{15} \log^2 n$ bytes, a close approximation of EpiGRAM's cost for $w = 128$.

Figure 12 clearly demonstrates EpiGRAM's low polylogarithmic scaling. Note that our communication grows slightly faster than the function $2^{15} \log^2 n$. This can be explained by the fact that we *fixed* a relatively low and constant word size $w = 128$; recall that to achieve $O(\log^2 n)$ scaling, we must choose $w = \Omega(\log^2 n)$. Still, our cost is closely modeled by $O(\log^2 n)$.

EpiGRAM is practical even for small n. The breakeven point with trivial GRAM (i.e., GRAM implemented by linear scans) is only $n = 512$ elements. Even non-garbled ORAMs have similar breakeven points. For example, Circuit ORAM [WCS15] gives the breakeven point $w = 128, n = 128$. At $n = 2^{20}$, EpiGRAM consumes ≈ 200× less communication than trivial GRAM.

Acknowledgements. This work was supported in part by NSF award #1909769, by a Facebook research award, a Cisco research award, and by Georgia Tech's IISP cybersecurity seed funding (CSF) award. This material is also based upon work supported in part by DARPA under Contract No. HR001120C0087. Work of the third author is supported in part by DARPA under Cooperative Agreement HR0011-20-2-0025, NSF grant CNS-2001096, US-Israel BSF grant 2015782, Google Faculty Award, JP Morgan Faculty Award, IBM Faculty Research Award, Xerox Faculty Research Award, OKAWA Foundation Research Award, B. John Garrick Foundation Award, Teradata Research Award, Lockheed-Martin Research Award and Sunday Group. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, the Department of Defense, or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright annotation therein.

References

[AKL$^+$20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 403–432. Springer, Heidelberg, May 2020.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[CCHR16] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled RAM or: How to delegate your database. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 61–90. Springer, Heidelberg, October / November 2016.

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

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the “free-XOR” technique. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, Heidelberg, March 2012.

[GHL$^+$14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 405–422. Springer, Heidelberg, May 2014.

[GKWY20] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy*, pages 825–841. IEEE Computer Society Press, May 2020.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In Venkatesan Guruswami, editor, *56th FOCS*, pages 210–229. IEEE Computer Society Press, October 2015.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 449–458. ACM Press, June 2015.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[GOS18] Sanjam Garg, Rafail Ostrovsky, and Akshayaram Srinivasan. Adaptive garbled RAM from laconic oblivious transfer. Cryptology ePrint Archive, Report 2018/549, 2018. `https://eprint.iacr.org/2018/549`.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Heidelberg, August 2018.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 719–734. Springer, Heidelberg, May 2013.

[LO17] Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 66–92. Springer, Heidelberg, August 2017.

[RR21] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. LNCS, pages 94–124. Springer, Heidelberg, 2021.

[RS19] Michael A. Raskin and Mark Simkin. Perfectly secure oblivious RAM with sublinear bandwidth overhead. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 537–563. Springer, Heidelberg, December 2019.

[SvS$^+$13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

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

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 850–861. ACM Press, October 2015.

[ZE13] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy*, pages 493–507. IEEE Computer Society Press, May 2013.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.

[ZWR$^+$16] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy*, pages 218–234. IEEE Computer Society Press, May 2016.

Supplementary Material

A Conservative Estimate of the Cost of Existing GRAM

Our objective here is to estimate the concrete cost of the best previously known realistic GRAM instantiation. Note that GRAM can always be instantiated by trivial linear scans, incurring $\approx 2 \cdot w \cdot n \cdot \kappa$ bits per access (when using [ZRE15] AND gates). Thus trivial GRAM has terrible scaling, but excellent constants. Of course, there is no point employing sophisticated methods if trivial GRAM is concretely the cheaper option.

From a concrete performance standpoint, [LO13] is the fastest prior nontrivial GRAM: subsequent GRAMs, aiming to resolve the circularity issue, use even more expensive tools, such as non-black-box public-key cryptography [GHL$^+$14] or extremely costly routing networks [GLO15]. Note, all prior GRAMs compose a (rough equivalent of our) leaky array with off-the-shelf ORAM, which incurs multiplicative overhead.

When instantiating GRAM based off of [LO13], we must choose an appropriate ORAM. Circuit ORAM [WCS15] is probably the best concrete choice. ORAMs with lower constants, e.g. square-root ORAM [RS19], have worse complexity. E.g., Circuit ORAM outperforms [ZWR$^+$16]'s square-root ORAM after around 2^{17} elements. For arrays smaller than 2^{17} elements, the [LO13]/square-root ORAM combination would not yet outperform trivial GRAM, due to the high constants imposed by [LO13]'s PRF use.

We thus use the combination of [LO13] and Circuit ORAM [WCS15] as a complete GRAM instantiation which reasonably reflects prior state of the art in concrete GRAM performance. We now conservatively estimate its cost and its break-even point with trivial GRAM.

Circuit ORAM includes recursively instantiated position maps. In our analysis, we ignore this recursion and conservatively only consider the Circuit ORAM's top-level tree. On a logical access, Circuit ORAM physically accesses two paths through its binary tree, each node of which has two elements. Thus, we can conservatively estimate the total number of physically accessed bits as:

$$4 \cdot w \cdot \log n$$

For word size $w = 128$ and $n = 2^{20}$, on a logical access, Circuit ORAM looks up a very modest (estimated) 10240 bits. However, under [LO13], this implies $2 \cdot 10240$ non-black-box calls to the PRF! If we instantiate the PRF using AES, which has 6400 AND gates, and we compare this to trivial linear scan on 2^{20} elements, we see that the costs are roughly the same ($\approx$ 4GB, assuming standard security parameter $\kappa = 128$, and 32 bytes per AND gate [ZRE15]). Thus, by the time a chosen RAM is large enough to warrant use of prior GRAM, each and every access (very) conservatively costs at least 4GB worth of communication.

B Language Translation Attempts

We examine three potential solutions to the language translation problem (Section 2.1) that do not help to achieve practical GRAM.

First, suppose we simply give to E each language X_i and the language Y. This is insecure: E can use the languages to decrypt wires and immediately learn some or all of G's input. It is safe to convey to E particular translation values between languages, but not the languages themselves.

Second, suppose G builds a simple encrypted table with n rows: the ith row holds the value $X_i \oplus Y$ encrypted by the GC labels that encode index i. At runtime, E jumps to row α, uses the GC encoding of α as a key to decrypt the row, and obtains the proper translation $X_\alpha \oplus Y$. Unfortunately, this is a linear cost construction and hence is too expensive. However, this attempt shows that it *is possible* to dynamically convey to E particular translation values.

Third, suppose each language X_i is not chosen uniformly, but rather is chosen deterministically by a PRF: let each $X_i \triangleq F_k(i)$. It is not safe to give the PRF key k to E, but we *can* place the key k *inside the GC*. From here, given a GC encoding of α, we can compute $F_k(\alpha) \oplus Y$ inside the GC and reveal the result to E, giving her the needed translation value. This technique is, in fact, sublinear, and is the main idea of the original garbled RAM construction [LO13]. Unfortunately, this approach has two serious problems:

- First, the construction is not practical. Evaluating a PRF inside the GC is extremely expensive. If we implement the PRF with AES and use 128-bit GC labels, then moving a single bit from RAM storage into the GC requires a full AES-128 evaluation, costing 6400 AND gates. If we use the best GC AND gate implementation [ZRE15], then each AND gate costs 32B, so G must send to E a staggering 200KB GC just to move one bit.
- Second, there is a circularity concern that arises in the security proof due to storing the PRF key k inside the GC *and* using $F_k(i)$ as a language for the same circuit. It is not known how to reduce this circularity to a simple assumption: currently, we must essentially assume that the entire construction is secure [LO13]. This problem can be removed by replacing the PRF by even more expensive techniques [GHL$^+$14,GLOS15].

However, this third approach does suggest a crucial idea: we can store and manipulate GC languages *inside the GC itself.* Our GRAM builds extensively uses this idea.

C Related Work – Extended

Practical Garbled Circuits. Since Yao originally described GC, a number of works improved the technique. This line of work improved the cost of Boolean gates such that it is now practical to run GC even for enormous circuits. The relevant work in this line is [ZRE15]: they gave an AND gate construction that consumes only one ciphertext if E knows the left AND gate input. We generalize

- Input:
  - A length RAM size n.
  - A bit width for RAM entries w.
- Output:
  - A sequence of languages $\mathcal{X}$. $\mathcal{X}$ stores $O(n \log n)$ languages each of length w.
  - The storage metadata $\mathcal{M}_{p \in [2n]}$ corresponding to each one-time index.
  - $n+1$ uniform permutations $\pi_{i \in [n+1]}$ to apply to physical storage.
- Procedure *G-schedule*(n, w):
  - For brevity and because it is computed locally by G, we do not explicitly list the *G-schedule* procedure. The high level idea is that G uniformly samples each π_i in his head, then uses these to track which one-time index is stored where and with which language (drawn from $\mathcal{X}$). This tracking allows G to assemble the storage metadata for each one-time index.

Fig. 13. *G-schedule* is a helper procedure that describes how G chooses all of the metadata he needs to garble the a GRAM.

this and show that the right input can be an *arbitrary* value, not just an encoding of zero or one. This simple generalization is crucial for efficiently manipulating languages inside the GC.

Oblivious RAM. Oblivious RAM [GO96]is a vibrant and important area of cryptographic research. It is well known that ORAM incurs $\Omega(\log n)$ bandwidth overhead [LN18]. This bound is tight: [AKL$^+$20] gave the first $O(\log n)$ ORAM that works for all block sizes.

The ORAM literature is vast. Techniques include (1) tree-based ORAMs, e.g. [SvS$^+$13,WCS15](2) hierarchical ORAMs, (3) square-root ORAMs, e.g. [RS19], (4) ORAMs specialized for MPC, and much more.

We technically present a new ORAM, though ours is specialized for GRAM. The key property of our ORAM is its highly deterministic read/write order, which makes it easy for G to predict most GC languages (see Section 2.4).

D Approach – Extended

In this appendix, we formalize GRAM helper procedures that were deferred from the main body. Specifically:

- Figure 13 formalizes the interface to *G-schedule*. This procedure formalizes how G chooses his random permutations and GC languages.
- Figure 14 formalizes *shuffle*. *shuffle* formalizes how uniform permutations are applied to the levels of RAM storage.
- Figure 15 formalizes the *hide* procedure. *hide* is used on each access to account for the single dummy that G chooses but that E does not read.

- Input:
    - A length n array $\mathit{Array}(x_0, ..., x_{n-1})$.
- Output:
    - The array $\mathit{Array}(x_0, ..., x_{n-1})$ where some levels of physical storage have been shuffled.
- Procedure *shuffle*$(\mathit{Array}(x_{i\in[n]}))$:
    - Let $2^{i+1} \leq 2n$ be the highest power of two that divides the RAM time T.
    - Let π_T be G's T-th chosen permutation (Figure 13).
    - Parties concatenate together each storage level $\ell \in [i+1]$. If $i = -1$ this concatenation is empty. They concatenate this with the stash. Let $\{\!\!\{\mathit{to\text{-}permute}\}\!\!\}$ denote the concatenated garbling. π_T is a permutation on 2^{i+2} elements.
    - Parties compute:

      $$\{\!\!\{\mathit{permuted}\}\!\!\} \leftarrow G\text{-}\mathit{permute}(\pi_T, \{\!\!\{\mathit{to\text{-}permute}\}\!\!\})$$

    - G splits off the first 2^{i+2} languages from $\mathcal{X}$: $(\mathit{target\text{-}languages}, \mathcal{X}) \leftarrow \mathcal{X}$ Parties trivially hold $[\![\mathit{target\text{-}languages}]\!]$.
    - Parties compute $\{\!\!\{\mathit{permuted}\}\!\!\} \oplus [\![\mathit{target\text{-}languages}]\!]$, then G opens his shares, ensuring the parties now hold a new garbling $\{\!\!\{\mathit{permuted}\}\!\!\}'$ where the chosen languages are stored in the leaves of the lazy permutation network $\tilde{\pi}$.
    - Parties save $\{\!\!\{\mathit{permuted}'\}\!\!\}$ in level $i + 1$ of physical storage.

Fig. 14. *shuffle* is a helper procedure that describes how G applies uniform permutation networks to the levels of physical storage.

E Encoding and Decoding

In this appendix we complete the EpiGRAM garbling scheme by specifying *En* and *De*; see Figure 16.

F Security Proofs

In this appendix, we prove EpiGRAM (Construction 1) correct and secure.

Our security proofs assume the existence of a circular correlation robust hash function H. We use the definition given by [ZRE15]:

Definition 5 (Circular Correlation Robustness). *We define two oracles:*

- $\mathit{circ}_\Delta(x, i, b) \triangleq H(x \oplus \Delta, i) \oplus b\Delta$ *where* $\Delta \in \{0, 1\}^{\kappa-1}1$.
- $\mathcal{R}(x, i, b)$ *is a random function with* κ*-bit output.*

A sequence of oracle queries (x, i, b) *is* legal *when the same value* (x, i) *is never queried with different values of* b. H *is circular correlation robust if for all poly-time adversaries* $\mathcal{A}$*:*

$$\left| \Pr_{\Delta}\left[\mathcal{A}^{\mathit{circ}_\Delta}(1^\kappa) = 1\right] - \Pr_{\mathcal{R}}\left[\mathcal{A}^{\mathcal{R}}(1^\kappa) = 1\right] \right| \textit{ is negligible.}$$

- INPUT:
  - For each populated level of physical storage i, G inputs $@'_i$: a physical address that holds a dummy.
  - For each address $@'_i$, G inputs D_i, the language for that physical address.
  - $\{\!|@|\!\}$, a garbling of the physical address of the accessed RAM element.
- OUTPUT:
  - Let j denote the storage level that holds address $@$.
  - For each populated level of physical storage i, parties output $\{\!|@_i|\!\}$, a physical address on that level. In particular, $@_j = @$ and for each $i \neq j$, $@_i = @'_i$.
  - Parties output $\llbracket D_j \rrbracket$, a sharing of language of the unaccessed dummy.
- PROCEDURE $hide(@'_i, D_i, \{\!|@|\!\})$:
  - For each populated level i, parties compute $\{\!|here_i|\!\}$ by comparing $@$ to two constants that indicate the highest and lowest address on level i.
  - For each populated level i, the parties compute and output an address:

$$\{\!|@_i|\!\} \triangleq \begin{cases} \{\!|@|\!\} & \text{if } here_i = 1 \\ \{\!|@'_i|\!\} & \text{otherwise} \end{cases}$$

  - Parties set $\llbracket D \rrbracket \leftarrow \llbracket 0 \rrbracket$.
  - For each populated level i, the parties update $\llbracket D \rrbracket$:

$$\begin{aligned} \llbracket D \rrbracket \leftarrow \llbracket D \rrbracket \oplus (\{\!|here_i|\!\} \cdot D_i) &= \llbracket D \rrbracket \oplus \llbracket here_i \cdot D_i \rrbracket && \text{Section 5.1} \\ &= \begin{cases} \llbracket 0 \rrbracket \oplus \llbracket D_i \rrbracket & \text{if } here_i = 1 \\ \llbracket D \rrbracket & \text{otherwise} \end{cases} && \text{only one bit } here_i \text{ is 1} \end{aligned}$$

  - Parties output $\llbracket D \rrbracket$.

Fig. 15. When E accesses physical storage, we ensure that she accesses an element on each nonempty level of storage. This prevents E from learning which level of storage holds the accessed element. This *hide* procedure accounts for the single dummy element that E *does not* read from storage (see Section 2.4).

EPIGRAM is correct and secure under the [BHR12] garbling scheme definitions. We define and then prove that EPIGRAM satisfies each of [BHR12]'s formal properties: *correctness*, *obliviousness*, *privacy*, and *authenticity*.

Definition 6 (Scheme Correctness). *A garbling scheme is* correct *if for all circuits $\mathcal{C}$ and all input strings x of appropriate length:*

$$De(d, Ev(\mathcal{C}, \tilde{\mathcal{C}}, En(e, x))) = \mathcal{C}(x) \qquad \textit{where } (\tilde{\mathcal{C}}, e, d) \leftarrow Gb(1^\kappa, \mathcal{C})$$

Correctness ensures the scheme properly implements circuits.

Theorem 2 (Correctness). EPIGRAM *(Construction 1) is* correct *(Definition 6).*

Proof. Our proof of correctness technically proceeds gate-by-gate through circuit $\mathcal{C}$. However, the details of this gate-by-gate handling, particularly for AND

- $En(e, x)$:
  - Let $n = |x|$ be the length of the input string.
  - Let $(e_0, ..., e_{n-1}) = e$ and suppose each $e_i \in \{0,1\}^\kappa$ is a uniform string.
  - For each $i \in [n]$, En computes and outputs $e_i \oplus x_i \cdot \Delta$.
- $De(d, \{\!\!\{x\}\!\!\})$:
  - Let $n = |x|$ be the length of the output string to decode.
  - For each $i \in [n]$, let $(d_i^0, d_i^1) = d_i$ denote two labels that respectively indicate an output zero/one.
  - For each $i \in [n]$, let $\langle \cdot, X_i \rangle = \{\!\!\{x_i\}\!\!\}$ denote E's output share.
  - For each $i \in [n]$, De selects a fresh nonce j, then computes and outputs:

$$\begin{cases} 0 & \text{if } H(X_i, j) = d_i^0 \\ 1 & \text{if } H(X_i, j) = d_i^1 \\ \bot & \text{otherwise} \end{cases}$$

Fig. 16. Our garbling scheme procedures *En* and *De*. The procedures are simple maps between cleartext and encoded values. The most interesting detail is in *De*. We use Free XOR-based labels [KS08], so each output label is either a string X_i or $X_i \oplus \Delta$. However, for privacy (Definition 9) it is crucial that the output decoding string d not reveal Δ. *De* breaks the correlation between labels by applying H.

and XOR gates, are simple and well known [ZRE15]. Thus, we focus on array initialization and array accesses.

Formally, we define the following *validity* property:

Definition 7 (RAM Validity). *An array* $Array(x_0, ..., x_{n-1})$ *is* valid *if:*

$$\mathit{flush}(Array(x_0, ..., x_{n-1})) = \{\!\!\{x_0, ..., x_{n-1}\}\!\!\}$$

I.e., validity ensures that we can recover the garbled content of a RAM. We argue that (1) *array-init* yields a valid array and (2) given a valid input array, *access* yields $\{\!\!\{x_\alpha\}\!\!\}$ and a valid output array. Since Sections 2 and 5 discuss the correctness of our approach at great length, we highlight formal details of these two steps only:

- The proof for the access procedure proceeds by induction on the index map: i.e., the top-level array is correct given that the index map is correct. The inductive argument is sensible because the bottom-most index map is instantiated by simple linear scans, and hence is trivially correct.
- *array-init* sets up validity by properly storing indices $0, ..., n-1$ in the index map. I.e., each index i is stored in one-time index i.
- Validity ensures array accesses will succeed by guaranteeing that there is be a path through the lazy permutation network to the needed one-time index; if there were not, we would be unable to flush. Validity is maintained because we write back to a fresh one-time index and appropriately update the index map.

- We ensure the lazy permutation network can always route at least n more values (until we refresh the RAM). This ensures we can always look up all n values in RAM.

We refer the reader back to Sections 2 and 5 for further discussion on the correctness of our RAM.

Because all four kinds of gates are correct, EpiGRAM is correct. □

Definition 8 (Scheme Obliviousness). *A garbling scheme is* oblivious *if there exists a simulator $\mathcal{S}_{obv}$ such that for any circuit $\mathcal{C}$ and all inputs x of appropriate length, the following are indistinguishable:*

$$(\tilde{\mathcal{C}}, En(e, x)) \overset{c}{=} \mathcal{S}_{obv}(1^\kappa, \mathcal{C}) \qquad \textit{where } (\tilde{\mathcal{C}}, e, \cdot) \leftarrow Gb(1^\kappa, \mathcal{C})$$

Obliviousness ensures that the GC and encoded inputs leak nothing to E.

Theorem 3 (Obliviousness). *If H is a circular correlation robust hash function, then* EpiGRAM *(Construction 1) is* oblivious *(Definition 8).*

Proof. By construction of a simulator $\mathcal{S}_{obv}$.

At a high level, our proof demonstrates two crucial facts:

1. We reveal to E permuted one-time indices $\pi(p)$ and we reveal physical addresses $@_i$. However, G applies uniform permutations to these values, so each is easily simulated.
2. G opens various sharings to E. We are careful that whenever G opens such a value, G's transmitted share is itself masked by a uniform string that is independent of all other openings. Hence, we can simulate each opening with a uniform string.

The remaining proof proceeds in detail.

Our proof approach. We modularly build up our obliviousness simulator from per-component simulators. Each simulator takes as input an input encoding and yields as output (1) a simulated garbled circuit and (2) simulated encoded output. These two simulated values are then argued indistinguishable from (1) the actual garbled circuit for the subcomponent and (2) the actual encoded output given by the subcomponent.

Usually, simulators take as input the real world output. The simulators for our intermediate procedures do not need to do this since we know the output distributions of our subcomponents precisely. Hence our simulators can simulate the output of our subcomponents. This leads to simpler hybrid arguments, since we can one-for-one substitute calls to real subcomponents by their simulator without restructuring any code in the hybrids.

We note that, as a minor and convenient abuse of notation, our simulators do not explicitly return simulated garbled circuits. We do this because, in all cases, the individual messages that compose our GCs are simply concatenated together. Thus, we simply state how we simulate each new piece of GC material, and we simplify notation by not explicitly threading and concatenating together portions of the GC. In line with this fact, we sometimes say that a simulator sends a value to E. This corresponds to adding that value to E's simulated view.

Finally, we use sharing notation in our simulators. This said, we only simulate E's view, so each share $\llbracket x \rrbracket$ has an empty left hand component: i.e., $\llbracket x \rrbracket = \langle \cdot, X \rangle$. This use of sharing notation is meant to clearly show the relationship between the simulator and the procedure it simulates. We intend for the reader to inspect each explicit simulator alongside the procedure it simulates.

Boolean circuit simulation. We did not formally specify Boolean AND gates in this work, instead delegating that concern to the [ZRE15] construction. We use their simulator to simulate AND gates. Hence, we can simulate GC evaluation of arbitrary Boolean circuits consisting of AND and XOR gates (XOR gates are locally computed and are trivially simulated [KS08]).

Sharing scale simulator. We construct a simulator for our scaling procedure (Figure 3). We prove security when G's share of the vector $\llbracket y \rrbracket$ is either (1) a uniform bitstring Y or (2) a bitstring $z\Delta$ for $z \in \{0,1\}$. The latter case arises when G introduces a garbled input.

– Simulator $\mathcal{S}_{scale}(\{\!\!\{ x^E \}\!\!\}, \llbracket y \rrbracket)$:
  - Let $\langle \cdot, X' \rangle = \{\!\!\{ x^E \}\!\!\}$ and let $\langle \cdot, Y' \rangle = \llbracket y \rrbracket$.
  - Let i be the gate-specific nonce.
  - Simulate *row* by uniformly sampling $r \in_{\$} \{0,1\}^{\kappa}$ then computing $\mathit{row}' \triangleq r \oplus H(X', i)$. This is indistinguishable from the real row. We prove this by two cases, depending on G's share of $\llbracket y \rrbracket$. If Y is uniform, then:

$$\begin{aligned} \mathit{row}' &= r \oplus H(X', i) \\ &= (r \oplus Y) \oplus H(X', i) \oplus Y \\ &\overset{c}{=} \mathcal{R}(X', i, 0) \oplus H(X', i) \oplus Y && \mathcal{R} \text{ is a random function} \\ &\overset{c}{=} \mathit{circ}_{\Delta}(X', i, 0) \oplus H(X', i) \oplus Y && \text{Definition 5} \\ &= H(X' \oplus \Delta, i) \oplus H(X', i) \oplus Y = \mathit{row} \end{aligned}$$

    Otherwise, if G's share of $\llbracket y \rrbracket$ is a string $z\Delta$ for $z \in \{0,1\}$ then:

$$\begin{aligned} \mathit{row}' &= r \oplus H(X', i) \\ &\overset{c}{=} \mathcal{R}(X', i, z) \oplus H(X', i) && \mathcal{R} \text{ is a random function} \\ &\overset{c}{=} \mathit{circ}_{\Delta}(X', i, z) \oplus H(X', i) && \text{Definition 5} \\ &= H(X' \oplus \Delta, i) \oplus z\Delta \oplus H(X', i) = \mathit{row} \end{aligned}$$

  - The simulator outputs $H(X', i) \oplus x \cdot (\mathit{row}' \oplus Y')$. Here, E's simulated share is indistinguishable from E's real output share by construction.

Pop-only stack simulators. Note that – due to space and because they are simple – we elided formal stack procedures *stack-init* and *pop* (Figure 4 lists the interface to these procedures). We similarly elide their simulators and instead simply claim that there exist simulators $\mathcal{S}_{\textit{stack-init}}$ and $\mathcal{S}_{\textit{pop}}$ that properly simulate E's view during these two procedures. Formally, both of these simulators simulate each of their gates, and the simulation is secure by a simple and unsurprising hybrid argument.

Lazy permutation network simulators. Next, we simulate E's view of our lazy permutation network.

We start by constructing simulators for inner and leaf nodes (Figures 5 and 6). Both *inner* and *leaf* are simple static circuits built from Boolean gates and Figure 3. Thus, we do not exhaustively list the simulators for these procedures. We do note one non-trivial detail: in both procedures, G opens a share to E. This is made simulatable by the fact that each nodes' input languages are chosen *uniformly*. Hence, G's opening can be simulated by uniform bits. Let $\mathcal{S}_{\textit{inner}}$ (resp. $\mathcal{S}_{\textit{leaf}}$) be the simulator for procedure *inner* (resp. *leaf*).

With simulators for the network nodes specified, we now construct simulators for the overall lazy permutation newtwork. We start by simulating the initialization of a network:

- Simulator $\mathcal{S}_{\tilde{\pi}\textit{-init}}(\cdot,\cdot)$:
  - Consider a full binary tree with n leaves.
  - For each node i in level k of the tree, trivially instantiate E's share of 2^k languages of appropriate length $[\![L_i^{j\in[2^k]}]\!]$. I.e., each language $L_i^j \triangleq \langle\cdot,0\rangle$.
  - For each internal node i on level k of the tree, simulate the initialization of two stacks (Figure 4):

$$s_i^\ell \triangleq \mathcal{S}_{\textit{init-stack}}([\![L_{2i}^{j\in[2^{k-1}]}]\!]) \qquad s_i^r \triangleq \mathcal{S}_{\textit{init-stack}}([\![L_{2i+1}^{j\in[2^{k-1}]}]\!])$$

  - Output E's simulated share of the lazy permutation network:

$$\tilde{\pi} = \left([\![L_0^{j\in[n]}]\!], (s^\ell_{i\in[n-1]}, s^r_{i\in[n-1]})\right)$$

The above simulation is indistinguishable from real by a trivial hybrid argument. Note that we defer simulation of the GCs for each of permutation nodes until we actually route the inputs:

- Simulator $\mathcal{S}_{\textit{route}}(\tilde{\pi}, \{\!\!\{\alpha^E\}\!\!\}, \{\!\!\{x\}\!\!\})$:
  - Let v denote the number of times $\tilde{\pi}$ has already been used.
  - Parse the lazy permutation into its parts:

$$\left([\![L_0^{j\in[n]}]\!], (s^\ell_{i\in[n-1]}, s^r_{i\in[n-1]})\right) = \tilde{\pi}$$

  - Trivially construct E's share of uniform language $[\![Y]\!] = \langle\cdot,0\rangle$.
  - Collect $[\![m]\!] \triangleq \{\!\!\{\alpha\}\!\!\}, \{\!\!\{x\}\!\!\}, [\![Y]\!]$.

- Simulate the opening of G's share by sampling a uniform string $row \in \{0,1\}^{|m|}$. Note that this is indistinguishable from real because L_0^v is an independently sampled uniform value that is unknown to E.
- Set $[\![m']\!] \leftarrow [\![m]\!] \oplus [\![L_0^v]\!] \oplus row$. I.e., $[\![m']\!]$ is simulated input to the first internal node.
- Traverse the tree from root to leaf α. At each internal leaf i, simulate the internal node procedure by invoking:
$$([\![m']\!], s_i^\ell, s_i^r) \leftarrow \mathcal{S}_{inner}(s_i^\ell, s_i^r, [\![m']\!], \alpha_i)$$
- Simulate the leaf by invoking $\mathcal{S}_{leaf}(\mathcal{C}_\alpha, [\![m']\!])$; output the result and the updated lazy permutation.
- To match the real world arrangement of GCs, the simulator rearranges the simulated GCs according to node ids.

The above simulator is indistinguishable from real by a simple hybrid argument. Note that $\mathcal{S}_{route}$ assumes that α is part of E's cleartext input. This is consistent with the fact that our lazy permutation network leaks values to E. We postpone simulating RAM indices to the simulation of our top level GRAM.

We note a tedious but important detail regarding the simulation of our lazy permutation network. In our simulation, we postponed the simulation of the node GCs until $\mathcal{S}_{route}$. This is sensible, because the moment when E calls *route* is the moment when she has the most information that could help her to distinguish the simulation from real. I.e., she holds an input to the root of the lazy permutation.

While this choice is natural, it has a problem. Suppose that a GC program uses a lazy permutation network of size n, but routes fewer than n inputs through the network. This can occur, e.g., when a GRAM of size n is accessed a number of times that is not a multiple of n. In such cases, there will be a number of GCs in the lazy permutation network that are not yet simulated. Thus, we must separately simulate the unused GCs in the permutation network. We simply mention this and do not fully flesh out such a simulator; we can clearly simulate node GCs where E does not receive input, since we can simulate the GCs even when E *does* receive input.

GRAM simulators. Now that we have constructed simulators for the lazy permutation network, we move on to our GRAM procedures. We start with simulators for the helper procedures (Figures 11 and 13 to 15):

- *G-schedule* (Figure 13) is local to G. We need not simulate.
- *shuffle* is easily simulated. First, $\mathcal{S}_{shuffle}$ simulates the call to *G-permute* by simulating the permutation network; This is done by simulating each of the constituent Boolean gates. Then, $\mathcal{S}_{shuffle}$ simulates G opening his share of the output. This is simulatable by a uniform string because *target-languages* is a uniform string chosen by G and independent of all other messages.
- We for now postpone discussion of *flush* (Figure 11).
- *hide* (Figure 15) is a simple circuit built from other gadgets, and so $\mathcal{S}_{hide}$ is simply constructed by simulating each of the constituent gates.

With these set, we focus on simulating GRAM initialization (Figure 9) and access (Figure 10). Simulating initialization is straightforward: $\mathcal{S}_{\mathit{array\text{-}init}}$ first simulates the initialization $\tilde{\pi}$ by calling $\mathcal{S}_{\tilde{\pi}\text{-}\mathit{init}}$. Then it recursively simulates initialization of the index map.

Array access is more detailed, and must handle important revelations to E. We fully formalize this simulator:

- Simulator $\mathcal{S}_{\mathit{access}}(A, \{\!\!\{\alpha\}\!\!\}, \{\!\!\{y\}\!\!\}, \{\!\!\{r\}\!\!\})$:
  - Simulate permutation of levels of storage: $A \leftarrow \mathcal{S}_{\mathit{shuffle}}(A)$
  - If $T = 2n$ reinitialize and try again, returning that result:

    $$\mathcal{S}_{\mathit{access}}(\mathcal{S}_{\mathit{array\text{-}init}}(\mathcal{S}_{\mathit{flush}}(A)), \{\!\!\{\alpha\}\!\!\}, \{\!\!\{y\}\!\!\}, \{\!\!\{r\}\!\!\})$$

    Otherwise, continue as follows:
  - Recursively simulate access to the index map:

    $$\{\!\!\{\pi(p)\}\!\!\} \leftarrow \mathcal{S}_{\mathit{access}}(\mathit{index\text{-}map}, \{\!\!\{\alpha\}\!\!\}, \{\!\!\{\pi(T)\}\!\!\}, \{\!\!\{0\}\!\!\})$$

  - Simulate G's opening of $\pi(p)$. **This is one of the most important points of our proof.** Let $\langle \cdot, P\rangle = \llbracket \pi(p) \rrbracket = \mathit{lsb}(\{\!\!\{\pi(p)\}\!\!\})$. The simulator uniformly samples a value $R \in [2n]$ *without replacement.* I.e., each time the simulator reaches this point it ensures that it samples a fresh value. (After array reinitialization, the simulator forgets which values it has shown to E; this allows us to simulate more than n accesses.) The simulator sends to E $R \oplus P$, revealing the value R. This is indistinguishable from real: in the real world, E views a value $\pi(p)$, but π is a *uniform permutation* and each p over the course of n accesses is distinct. Hence, each such $\pi(p)$ appears uniformly chosen without replacement.
  - Simulate routing of the permutation network.

    $$(\{\!\!\{@\}\!\!\}, \llbracket X \rrbracket) \leftarrow \mathcal{S}_{\mathit{route}}(\tilde{\pi}, \{\!\!\{\pi(p)\}\!\!\}, T)$$

  - Simulate the *hide* procedure (Figure 15):

    $$(\{\!\!\{@_i\}\!\!\}, \llbracket D_j \rrbracket) \leftarrow \mathcal{S}_{\mathit{hide}}(@'_i, D_i, \{\!\!\{@\}\!\!\})$$

  - Simulate G's opening of each physical address on each populated level. **This is one of the most important points of our proof.** For each populated level i let $\langle \cdot, A_i\rangle = \llbracket @_i \rrbracket = \mathit{lsb}(\{\!\!\{@_i\}\!\!\})$. For each such level, the simulator uniformly samples a value $R_i \in [2^{i+1}]$ *without replacement.* I.e., the simulator never reveals to E the same physical address more than once. (After a level is permuted, the simulator forgets which addresses it revealed on that level.) The simulator sends to E $A_i \oplus R_i$, revealing to E the value R_i. This is indistinguishable from real: Recall that the levels of RAM are *permuted* according to uniform permutations $\pi_0, ..., \pi_n$. Hence, each level i is uniformly shuffled. Since all 2^{i+1} elements are uniformly shuffled, the real value $@_i$ is indistinguishable from a uniformly sampled (without replacement) index.

- Read each simulated address R_i and XOR the values together. XOR with this result the values $[\![D_j]\!]$ and $[\![X]\!]$. Let $\{\!|x_\alpha|\!\}$ denote the reslut. Output $\{\!|x_\alpha|\!\}$.
- Simulate the Boolean circuit $\{\!|r \cdot x_\alpha \oplus \bar{r} \cdot y|\!\}$ and place the result in the first slot of the stash. Place the trivial share of $\{\!|0|\!\}$, a fresh dummy, in the second slot of the stash.
- Increment the timer: $T \leftarrow T + 1$.

As a final detail, we now revisit the simulator $\mathcal{S}_{\mathit{flush}}$. Just like the above $\mathcal{S}_{access}$ simulator, $\mathcal{S}_{\mathit{flush}}$ must take care when revealing physical addresses to E. But using the same argument as above, the *flush* simulator can just choose locations uniformly without replacement and reveal these to E.

We reiterate the crucial points of the above simulation: during an array access, E views a permuted one-time index $\pi(p)$ and $O(\log n)$ physical addresses. However, all of these values are masked by uniform permutations chosen by G, and hence can be simulated by uniformly sampling indices without replacement. Other than this, the simulation is a straightforward reduction to simulators of the subcomponents. The full simulation is indistinguishable from real by a straightforward and unsurprising hybrid argument.

Top level simulator. We have now proved the GRAM simulators secure. We use these simulators to simulate our top level obliviousness simulator $\mathcal{S}_{obv}$:

– Simulator $\mathcal{S}_{obv}(1^\kappa, \mathcal{C})$:
  - Simulate each bit of the input with a uniform string. This is indistinguishable from real since the real garbling procedure chooses the zero encoding of each input string uniformly.
  - Step through the circuit gate by gate. Handle each gate by calling the appropriate simulator and placing the result on the gate's output wire:
    - ∗ XOR gate: XOR the two simulated inputs.
    - ∗ AND gate: call [ZRE15]'s AND gate simulator.
    - ∗ ARRAY gate: call $\mathcal{S}_{\mathit{array\text{-}init}}$.
    - ∗ ACCESS gate: call $\mathcal{S}_{access}$.

This simulation is indistinguishable from real by the security of each of the invoked gate simulators and by a straightforward hybrid argument.

EpiGRAM is oblivious. □

Definition 9 (Scheme Privacy). *A garbling scheme is* private *if there exists a simulator $\mathcal{S}_{prv}$ such that for any circuit $\mathcal{C}$ and all inputs x of appropriate length, the following are computationally indistinguishable:*

$$(\tilde{\mathcal{C}}, En(e, x), d) \overset{c}{=} \mathcal{S}_{prv}(1^\kappa, \mathcal{C}, \mathcal{C}(x)) \qquad \textit{where } (\tilde{\mathcal{C}}, e, d) \leftarrow Gb(1^\kappa, \mathcal{C})$$

Privacy ensures that the GC and encoded inputs together with the output decoding string reveal nothing beyond the output.

Theorem 4 (Privacy). *If H is a circular correlation robust hash function, then* EpiGRAM *(Construction 1) is* private *(Definition 9).*

Proof. By construction of a privacy simulator $\mathcal{S}_{prv}$. Privacy follows from obliviousness (Theorem 3) and the definition of De (Figure 16).

The privacy simulator (1) simulates a GC and encoded input by calling the obliviousness simulator, (2) evaluates the simulated GC on the simulated input to obtain encoded output, and (3) forges an output decoding string that ensures the encoded output decodes to the correct cleartext value:

- Simulator $\mathcal{S}_{prv}(1^\kappa, \mathcal{C}, y)$:
  - Compute $(\tilde{\mathcal{C}}, \{\!|x|\!\}) \leftarrow \mathcal{S}_{obv}(1^\kappa, \mathcal{C})$.
  - Compute $\{\!|y|\!\} \leftarrow Ev(\mathcal{C}, \tilde{\mathcal{C}}, \{\!|x|\!\})$.
  - Let $n = |y|$. Initialize a length-n decoding string d'.
  - For each $i \in [n]$, let $\langle \cdot, Y_i \rangle = \{\!|y_i|\!\}$ denote E's simulated output.
  - For each $i \in [n]$, let j denote the nonce used in De and let $r_i \in_\$ \{0,1\}^\kappa$ be a uniform string. Assign the ith index of the decoding string as follows:
$$d'_i \leftarrow \begin{cases} (H(Y_i, j), r_i) & \text{if } y_i = 0 \\ (r_i, H(Y_i, j)) & \text{otherwise} \end{cases}$$
  - Output $(\tilde{\mathcal{C}}, \{\!|x|\!\}, d')$.

We argue:

$$(\tilde{\mathcal{C}}, En(e, x), d) \overset{c}{=} \mathcal{S}_{prv}(1^\kappa, \mathcal{C}, \mathcal{C}(x)) \qquad \text{where } (\tilde{\mathcal{C}}, e, d) \leftarrow Gb(1^\kappa, \mathcal{C})$$

First, note that the simulation correctly outputs y: the forged decoding string d' is precisely chosen such that this holds. Second, note that each entry d'_i is indistinguishable from real. The real entry d_i is as follows:

$$(H(Y_i, j), H(Y_i \oplus \Delta, j))$$

Note the following indistinguishability argument:

$$\begin{aligned}
& d_i = (H(Y_i, j), H(Y_i \oplus \Delta, j)) \\
&= \begin{cases} (H(Y_i, j), H(Y_i \oplus \Delta, j)) & \text{if } y_i = 0 \\ (H(Y_i, j), H(Y_i \oplus \Delta, j)) & \text{otherwise} \end{cases} \\
&= \begin{cases} (H(Y_i, j), circ_\Delta(Y_i, j, 0)) & \text{if } y_i = 0 \\ (circ_\Delta(Y_i \oplus \Delta, j, 0), H(Y_i \oplus \Delta, j)) & \text{otherwise} \end{cases} && \text{Definition 5} \\
&\overset{c}{=} \begin{cases} (H(Y_i, j), \mathcal{R}(Y_i, j, 0)) & \text{if } y_i = 0 \\ (\mathcal{R}(Y_i \oplus \Delta, j, 0), H(Y_i \oplus \Delta, j)) & \text{otherwise} \end{cases} && \text{Definition 5} \\
&\overset{c}{=} \begin{cases} (H(Y_i, j), r_i) & \text{if } y_i = 0 \\ (r_i, H(Y_i \oplus \Delta, j)) & \text{otherwise} \end{cases} && \mathcal{R} \text{ is a random function} \\
&\overset{c}{=} d'_i
\end{aligned}$$

Because of the indistinguishability given by $\mathcal{S}_{obv}$ and because d is constructed using H, the joint distribution of GC, encoded input, and decoding string d is indistinguishable.

EpiGRAM is private. □

Definition 10 (Scheme Authenticity). *A garbling scheme is* authentic *if for all circuits $\mathcal{C}$, all inputs x of appropriate length, and all poly-time adversaries $\mathcal{A}$ the following probability is negligible in κ:*

$$\begin{aligned} &Pr(Y' \neq Ev(\mathcal{C}, \tilde{\mathcal{C}}, En(e, x)) \wedge De(d, Y') \neq \bot) \\ &\qquad \textit{where } (\tilde{\mathcal{C}}, e, d) \leftarrow Gb(1^\kappa, \mathcal{C}) \textit{ and where } Y' \leftarrow \mathcal{A}(\mathcal{C}, \tilde{\mathcal{C}}, En(e, x)) \end{aligned}$$

Authenticity ensures that even a malicious E cannot construct output labels that successfully decode except by running the GC as intended.

Theorem 5 (Authenticity). *If H is a circular correlation robust hash function, then* EpiGRAM *(Construction 1) is* authentic *(Definition 10).*

Proof. Authenticity holds by the definition of the privacy simulator (Theorem 4) and by our choice of De (Figure 16).

Recall that authenticity allows $\mathcal{A}$ access to a garbled circuit $\tilde{\mathcal{C}}$ and encoded input $\{\!|x|\!\}$ produced by Gb. To derive a contradiction, let $(\tilde{\mathcal{C}}', \{\!|x|\!\}', d')$ be a garbling constructed by the privacy simulator $\mathcal{S}_{prv}$. Now, suppose $\mathcal{A}$ is instead given $(\tilde{\mathcal{C}}', \{\!|x|\!\}')$. Notice that here, it is infeasible for $\mathcal{A}$ to forge an encoded output Y' that successfully decodes. Indeed, suppose $\mathcal{A}$ is able to flip even a single bit y_i of the output. But by the definition of the privacy simulator, this would require that $\mathcal{A}$ guess a uniform value $r_i \in \{0,1\}^\kappa$ that was sampled by the simulator and that is independent of $\mathcal{A}$'s view, which is clearly infeasible. $\mathcal{A}$ cannot forge an output when given a simulated GC.

If $\mathcal{A}$ can forge an output when given a *real* GC, then we can construct a polytime privacy distinguisher:

- Distinguisher $\mathcal{D}^{\mathcal{C}}_{prv}(\tilde{\mathcal{C}}, X, d)$:
  - Compute $Y \triangleq Ev(\mathcal{C}, \tilde{\mathcal{C}}, X)$ to evaluate the GC normally.
  - Compute $Y' \triangleq \mathcal{A}(\mathcal{C}, \tilde{\mathcal{C}}, X)$ to forge an output.
  - Compute and output the following bit:

$$De(d, Y') \neq \bot \wedge De(d, Y') \neq De(d, Y)$$

Assume that $\mathcal{D}^{\mathcal{C}}_{prv}$ outputs 1 with non-negligible probability when given real-world input, corresponding to the fact that $\mathcal{A}$ can forge an output with non-negligible probability. Then $\mathcal{D}_{prv}$ is indeed a distinguisher, since we already concluded $\mathcal{A}$ cannot succeed (except with negligible probability) when given simulated input. But EpiGRAM is private, and hence no such distinguisher should exist. We have reached a contradiction. It must be that $\mathcal{A}$ cannot forge an output given a real-world input except with negligible probability.

EpiGRAM is authentic. □

G Efficiency Proofs

In this section we prove EpiGRAM achieves $O(\log^2 n)$ overhead. To prove this, we derive costs for the various components of our RAM. In particular, we:

1. Remind the reader of the cost of our scaling procedure (Figure 3).
2. Derive the cost of stacks internal to our lazy permutation network.
3. Use the cost of stacks to derive the total cost of a lazy permutation network.
4. Derive the cost of all permutations applied to physical storage by G.
5. Derive the cost of the helper procedure *hide* (Figure 15).
6. Show that the index map, which is instantiated by a recursive chain of RAMs, incurs total $O(\log^4 n \cdot \kappa)$ amortized cost per access (if the index map stores entries of size $w = 2 \log n$).
7. Prove that, for $w = \Omega(\log^2 n)$, EpiGRAM incurs total $O(w \cdot \log^2 n \cdot \kappa)$ amortized cost per RAM access.

G.1 Costs of Subcomponents

We start by reminding the reader that our scaling procedure (Figure 3) avoids factor κ overhead:

Lemma 1 (Scaling Cost). *Let $\{\!|x^E|\!\}$ be a garbled bit and let $[\![y]\!]$ for $y \in \{0,1\}^\kappa$ be a shared vector. Parties compute $[\![x \cdot y]\!]$ (Figure 3) for κ bits of communication and $O(\kappa)$ computation.*

Proof. Trivial from Figure 3. G sends only a single length-κ string *row*. □

Based on the above lemma, we briefly observe that pop-only stacks consume amortized $O(\log n)$ overhead per pop:

Lemma 2 (Stack Cost). *Let $s = Stack(x_0, ..., x_{n-1})$ be a size-n stack (Construction 2) with w-bit entries. Let $m = O(n)$ be a number of pops linear in the stack size. For each $i \in [m]$ let $\{\!|p_i^E|\!\}$ be a garbled bit. Consider a sequence of m calls to pop:*

$$(\cdot, s) \leftarrow pop(s, \{\!|p_i^E|\!\})$$

The above calls incur total $O(w \cdot n \cdot \log n)$ communication and computation.

Proof. By analysis given by [ZE13] and because we replace AND gates – which have factor κ overhead – with our scaling gates – which do not (Lemma 1). □

Based off stack costs, we calculate the cost of our lazy permutation network. A fully routed lazy permutation network incurs $O(w \cdot n \cdot \log^2 n)$ cost:

Lemma 3 (Lazy Permutation Network Cost). *Let $\tilde{\pi}$ be a lazy permutation network on n elements where each leaf node p is configured by storage metadata (Definition 4) $\mathcal{M}_p$ with $O(\log n)$ entries each with language of width w. Let π*

be an arbitrary permutation on n elements. For each $i \in [n]$ let the parties hold $\{\!\!\{\pi(i)\}\!\!\}$. Suppose the parties fully route $\tilde{\pi}$. I.e., for each $i \in [n]$ they call:

$$(\cdot, \cdot, \tilde{\pi}) \leftarrow \mathit{route}(\tilde{\pi}, \{\!\!\{\pi(i)\}\!\!\}, 0)$$

If $w = \Omega(\log n \cdot \kappa)$ then the parties consume total $O(w \cdot n \cdot \log^2 n)$ communication and computation.

Proof. By totaling the cost of stacks in $\tilde{\pi}$.

First, we show that each leaf node costs only $O(w \cdot \log n + \log^2 n \cdot \kappa)$, and hence all leaf nodes together cost $O(w \cdot n \cdot \log n + n \log^2 n \cdot \kappa) = O(w \cdot n \cdot \log^2 n)$. Each leaf performs $O(\log n)$ comparisons on an integer of length $\log n$. Each integer comparison can be implemented using a circuit with $O(\log n)$ gates, hence total $O(\log^2 n \cdot \kappa)$ cost. With the comparisons computed, the leaf then computes $O(\log n)$ scalings, each incurring cost w.

Now, $\tilde{\pi}$ internally holds $2n - 2$ stacks, though these stacks decrease in size towards the leaves of the network. I.e., the network has $\log n$ levels, and each inner node on level i has two stacks of size 2^{i-1}. Recall from Lemma 3 that 2^i calls to *pop* on a stack with 2^{i-1} elements of width $O(w)$ costs total $O(w \cdot 2^i \cdot \log 2^i)$. For each of the 2^{i+1} stacks on level i, a fully utilized lazy permutation issues $2^{\log n - i}$ calls to *pop*. Thus we can sum up costs as follows:

$$\begin{aligned}
&\sum_{i=0}^{\log n-1} 2^{i+1} \cdot O\left(w \cdot 2^{\log n - i} \cdot \log 2^{\log n - i}\right) \\
&= O\left(\sum_{i=0}^{\log n-1} 2^{i+1} \cdot \left(w \cdot \frac{2^{\log n}}{2^i} \cdot \log\left(\frac{2^{\log n}}{2^i}\right)\right)\right) \\
&= O\left(\sum_{i=0}^{\log n-1} w \cdot n \cdot \log\left(\frac{n}{2^i}\right)\right) \\
&= O\left(w \cdot n \cdot \left(\sum_{i=0}^{\log n-1} \log n - i\right)\right) \\
&= O(w \cdot n \cdot \log^2 n)
\end{aligned}$$

The total costs of inner and leaf nodes therefore sum to $O(w \cdot n \cdot \log^2 n)$. □

Lemma 4 (Traditional Permutation Network Cost). *Let $(\pi_0, ..., \pi_n)$ be a sequence of $n+1$ permutations chosen by G-schedule(n, w) and let $\{\!\!\{x_0\}\!\!\}, ..., \{\!\!\{x_n\}\!\!\}$ be n garbled arrays such that each x_i has length appropriate for permutation π_i. Let each element of each array x_i have width w. Suppose the parties permute each array using G-permute (Figure 2):*

$$\{\!\!\{\pi(x_i)\}\!\!\} \leftarrow \textit{G-permute}(\pi_i, \{\!\!\{x_i\}\!\!\})$$

Then the parties use $O(w \cdot n \cdot \log^2 n \cdot \kappa)$ communication and computation.

Proof. By totalling the cost of each permutation network.

Recall that a permutation network on 2^i garbled elements each of width w incurs $O(w \cdot 2^i \log 2^i \cdot \kappa)$ cost (Figure 2). Recall also that for each $i \in [n]$, the procedure *G-schedule* samples a permutation of size $2k$ such that $k \leq n$ and such that k is the largest power of two that divides i. Additionally, *G-schedule* appends a final permutation of size $4n$.

Note that by the above strategy, for each $i \in [\log n]$ there are $2^{\log n - i - 1}$ permutations of size $2 \cdot 2^i$. Additionally, there is one permutation of size $2n$ and one of size $4n$. These two largest permutations have total cost $O(w \cdot n \cdot \log^2 n \cdot \kappa)$. We can summarize the costs of all smaller permutations as follows:

$$
\begin{aligned}
&\sum_{i=0}^{\log n - 1} 2^{\log n - i - 1} \cdot O(w \cdot 2^i \cdot \log 2^i \cdot \kappa) \\
&= O\left(w \cdot \kappa \cdot 2^{\log n} \cdot \left(\sum_{i=0}^{\log n - 1} \frac{i \cdot 2^i}{2^{i-1}} \right) \right) \\
&= O\left(w \cdot \kappa \cdot n \cdot \left(\sum_{i=0}^{\log n - 1} i \right) \right) \\
&= O(w \cdot n \cdot \log^2 n \cdot \kappa)
\end{aligned}
$$

The total cost of permutations assigned by *G-schedule* is $O(w \cdot n \cdot \log^2 n \cdot \kappa)$. □

Before we explore the amortized cost of RAM accesses, we quickly derive the cost of the *hide* helper procedure:

Lemma 5 (Hide Procedure Cost). *Let $@'_i$ be $O(\log n)$ physical addresses each of length $O(\log n)$ bits. Let D_i be $O(\log n)$ languages each of length w. Let $\lBrace @ \rBrace$ be a garbled physical address. Suppose the parties invoke hide:*

$$
\textit{hide}(@'_i, D_i, \lBrace @ \rBrace)
$$

If $w = \Omega(\kappa)$ then the parties consume total $O(w \cdot \log^2 n)$ communication and computation.

Proof. hide uses $O(\log n)$ integer comparisons for integers of size $O(\log n)$. Each integer comparison can be implemented using a circuit with $O(\log n)$ gates, hence total $O(\log^2 n \cdot \kappa) = O(w \cdot \log^2 n)$ cost. Additionally, *hide* involves $O(\log n)$ vector scalings, each of cost w. Hence total cost is bounded by $O(w \cdot \log^2 n)$. □

G.2 Costs of RAM

Now that we have derived the costs of the subcomponents of our RAM, we derive the amortized cost of our core *access* procedure.

Recall that the RAM recursively instantiates a *index map* which maps each logical index to a one-time index. Because of the recursive instantiation, we are at risk of incurring an additional factor $\log n$ overhead. To circumvent this, we use a trick given by [SvS$^+$13]: we instantiate the top level RAM with substantially wider entries than the index map. I.e., we store blocks of width $w = \Omega(\log^2 n)$ in the top level RAM and blocks of width $w = 2 \cdot (\log n + 1)$ in lower levels of RAM.

In practice, we play with constants for the top level RAM. For example, we store blocks of size, say 128, in the top level.

We show that the top-level index map has total cost $O(\log^4 n \cdot \kappa)$. Then, we show that the top level RAM has total cost $O(w \cdot \log^2 n \cdot \kappa)$.

Lemma 6 (Index Map Efficiency). *Let Array$(x_0, ..., x_{n-1})$ be a size-n array with entries of width $w = 2 \cdot (\log n + 1)$. Then each call to access (Figure 10) consumes amortized $O(\log^4 n \cdot \kappa)$ communication and computation.*

Proof. By amortizing the cost of the lazy permutation network (Lemma 3) and traditional permutations (Lemma 4).

n accesses to a size-n RAM together utilize:

- A size-$2n$ lazy permutation where the leaves store languages of size $w \cdot \kappa$.
- $n + 1$ traditional permutations.

By amortizing the costs of these components to each access, we see that each access incurs $O(w \cdot \log^2 n \cdot \kappa)$ cost. The *hide* procedure – which is called once per acces – also has cost bounded by $O(w \cdot \log^2 n \cdot \kappa)$ (Lemma 5).

Crucially, each RAM access requires exactly one recursive access to its index map. The index map for each level of RAM must uniquely identify one out of $2n$ one-time indices, and hence each we must look up an entry of size $\log n + 1$. Since we store $2(\log n + 1)$ bits per entry, each recursively instantiated RAM is at most half the size of its parent. Thus, we have at most $\log n$ levels of RAM (recall that the bottom-most level of RAM is instantiated by simple linear scans). Since the cost of each level of RAM is bounded by $O(w \cdot \log^2 n \cdot \kappa)$ and there are $O(\log n)$ levels of RAM, the total cost is $O(w \cdot \log^3 n \cdot \kappa) = O(\log^4 n \cdot \kappa)$. □

Theorem 6 (Access Efficiency). *Let Array$(x_0, ..., x_{n-1})$ be a size-n array with entries of width $w = \Omega(\log^2 n)$. Then each call to access (Figure 10) consumes amortized $O(w \cdot log^2 n \cdot \kappa)$ communication and computation.*

Proof. By amortizing the cost of the lazy permutation network (Lemma 3) and traditional permutations (Lemma 4) and because the index map has total cost $O(\log^4 n \cdot \kappa)$ (Lemma 6)

The proof is nearly identical to that of Lemma 6, except that we ignore recursive RAM instantiation since we have already proved the index map has cost $O(\log^4 n \cdot \kappa)$ per access.

EpiGRAM achieves $O(\log^2 n)$ overhead. □