Garbled RAM From One-Way Functions

Sanjam Garg* Steve Lu† Rafail Ostrovsky‡ Alessandra Scafuro§

Abstract

Yao's garbled circuit construction is a fundamental construction in cryptography and recent efficiency optimizations have brought it much closer to practice. However these constructions work only for circuits and garbling a RAM program involves the inefficient process of first converting it into a circuit. Towards the goal of avoiding this inefficiency, Lu and Ostrovsky (Eurocrypt 2013) introduced the notion of "garbled RAM" as a method to garble RAM programs directly. It can be seen as a RAM analogue of Yao's garbled circuits such that, the size of the garbled program and the time it takes to create and evaluate it, is proportional only to the running time on the RAM program rather than its circuit size.

Known realizations of this primitive, either need to rely on strong computational assumptions or do not achieve the aforementioned efficiency (Gentry, Halevi, Lu, Ostrovsky, Raykova and Wichs, EUROCRYPT 2014). In this paper we provide the first construction with strictly poly-logarithmic overhead in both space and time based only on the minimal and necessary assumption that one-way functions exist. Our scheme allows for garbling multiple programs being executed on a persistent database, and has the additional feature that the program garbling is decoupled from the database garbling. This allows a client to provide multiple garbled programs to the server as part of a pre-processing phase and then later determine the order and the inputs on which these programs are to be executed, doing work independent of the running times of the programs itself.

*University of California, Berkeley. Computer Science Division. sanjamg@berkeley.edu
†University of California, Los Angeles. Department of Computer Science. stevelu@cs.ucla.edu
‡University of California, Los Angeles. Department of Computer Science and Mathematics. rafail@cs.ucla.edu
§University of California, Los Angeles. Department of Computer Science. scafuro@cs.ucla.edu

1 Introduction

As individuals and organizations push massive amounts of personal data and the associated computational demands to the cloud, guaranteeing privacy of this data wile simultaneously enabling easy access to it poses tremendous challenges. Consider the following concrete problem. A client, say Alice, wants to store a large private dataset or database D on an untrusted server, referred to as "the cloud." Subsequently, Alice wants the cloud to be able to compute and learn the output of arbitrary dynamically chosen private programs $P_1, P_2, \ldots$ on private inputs $x_1, x_2, \ldots$ and the previously stored dataset, which gets updated as these programs are executed. Furthermore, Alice wants to achieve this non-interactively, i.e. by sending just one message for each program/input to the server.

Solutions using Yao's Garbled Circuits or Fully Homomorphic Encryption. The first feasibility solution for this problem was provided by Yao [Yao82]. However Yao's approach requires that the program be first converted to a circuit — the size of which will need to grow at least with the size of the input. Hence for each program that Alice wants the cloud to compute, it will need to send a message that grows with the size of the database. A potential attempt to mitigate this is to use Fully Homomorphic Encryption, first provided by Gentry [Gen09]. While this reduces the size of Alice's message, the cloud still needs to touch the entire encrypted database. Consequently the work of the cloud still grows with the size of the database. These solutions can be prohibitive for various applications, e.g. for binary search the size of the database can be exponentially larger than execution path of the insecure solution. We note that additionally even in settings where the size of the database is small, generic transformations from RAM programs with running time T result in a circuit of size $O(T^3 \log T)$ [CR73, PF79], which can be prohibitively inefficient.

Garbled RAMs. Motivated by the above considerations, Lu and Ostrovsky [LO13b] introduced the notion of garbled random-access machines (garbled RAMs) as a method to garble RAM programs directly. Garbled RAMs can be seen as a RAM analogue of Yao's garbled circuits, such that the size of the garbled program, the time it takes to create and evaluate it is proportional only to the running time on the RAM pogram (up to poly-logarithmic factors), rather than the size of its circuit representation.

In more detail, we will use the notation $P^D(x)$ to denote the execution of some RAM program P on input x with initial memory D. A garbled RAM scheme should provide a mechanism to garble the data D into garbled data $\tilde{D}$, the program P into garbled program $\tilde{P}$ and the input x into garbled input $\tilde{x}$ such that given $\tilde{D}, \tilde{P}$ and $\tilde{x}$ allows for computing $P^D(x)$ and nothing more. Furthermore, up to only poly-logarithmic factors in the running time of the RAM $P^D(x)$ and the size of D, we require that the size of garbled data $\tilde{D}$ is proportional only to the size of data D, the size of the garbled input $\tilde{x}$ is proportional only to that of x and the size and the evaluation time of the garbled program $\tilde{P}$ is proportional only to the running time of the RAM $P^D(x)$.

Both the original Lu and Ostrovsky [LO13b] construction and its follow up [GHRW14a, LO14, GHL+14] need to make strong computational assumptions for achieving the above described efficiency goals. In particular, the security of the original Lu-Ostrovsky relies on a somewhat complex "circular" assumption involving Yao garbled circuits and PRFs and the Gentry et al. [GHRW14a] construction uses identity-based encryption (IBE). Lu and Ostrovsky [LO14] describe an alternative construction based just on one-way functions but that does not meet the above mentioned efficiency goals. Specifically for this solution, the size of the garbled program grows by an additional factor of $|D|^\epsilon$ for a constant $\epsilon > 0$, which can be prohibitive for the envisioned applications. These works leave open the following problem.

Can we base security of poly-logarithmically efficient garbled RAMs just on the existence of one-way functions?

1.1 Our Results

In this paper we provide the first construction of an efficient garbled RAM that can be based on one-way functions alone, thus making it a suitable replacement for Yao's garbled circuits with the added benefits of being in the RAM model. We state this as our main theorem:

Main Theorem (Informal). *Assuming one-way functions, there exists a secure garbled RAM scheme, where the size of the garbled database is $|D| \cdot \mathsf{poly}(\kappa)$, size of the garbled input is $|x| \cdot \mathsf{poly}(\kappa)$ and the size of the garbled program and its evaluation time is $T \cdot \mathsf{poly}(\log T, \log |D|, \kappa)$ where T is the running time of program P. Here κ is the security parameter.*

Our contribution also includes an interesting strengthening of the garbled RAM definition where garbling the database is decoupled from the garbling of the program, and only tied together by the garbling of the input. Our constructions do achieve this stronger definition. We believe this will be of independent interest in various outsourcing computation applications. We provide a motivating example application to the pre-processing model below.

Persistent Garbled Database. Just as in previous works, our construction allows for execution of multiple programs on the garbled memory, such that the running time of the client and the server per program is proportional to the RAM runtime of that program. Furthermore the garbled database is persistent in the sense that the modifications made to it by the execution of garbled programs are maintained across different program executions. At the same time an attacker can not execute programs out of order, replay old garbled databases to new programs, or modifying the underlying contents of the garbled database beyond what is dictated by the GRAM evaluation algorithm.

Preprocessing Model. A nice feature of our solution is that the online work of a client can be further reduced when preprocessing is allowed. In more detail, consider a setting where in a preprocessing phase the client provides the cloud, not just a garbled database but also a set of garbled programs that she might want to execute in the future. Subsequently during the online phase she can dynamically decide the execution order of her previously garbled programs, and execute them on inputs of her choice. A nice feature of this solution is that the clients online overhead for executing a program reduces just to the size of the private inputs (with a $\mathsf{poly}(\kappa)$ overhead), completely independent of the size of the data and the running time of the programs. Technical this becomes possible because of the decoupling of data garbling from program garbling.

Input-specific running time. If one is willing to disclose the exact running time of a specific execution, then the running time of a garbled RAM computation can be made input specific which could be much faster than the worst-case running time. This was first explicitly explored by Goldwasser et al. [GKP+13] in the context of secure TM computation. This applies to a wide range of algorithms that have a gap between "typical"-case and worst-case time complexity, including randomized algorithms which could have exponential worst-case running time (Las Vegas), or heuristic algorithms that perform well in practice.

In our setting, the client can garble a sequence of generic CPU steps and provide them to the cloud. Later on, the client just needs to provide a short garbled input, which allows the cloud to do the computation in input specific running time. Interestingly this only consumes garbled CPU steps proportional to the input specific running time and the left over CPU steps can be used for executing the next program. In other words the clients overall work will be proportional only to the sum of the input-specific running times of the programs it executes with the server.

Output Privacy and Verifiability. We note that if in an application output privacy is desired then we can always make the garbled program perform an extra step of encrypting or authenticating the intended output before actually outputting it.

Secure RAM Computation. Much like how garbled circuits can be applied to secure circuit computation, so can garbled RAM be used for secure RAM computation. This allows us to perform one-round secure RAM computation in the OT-hybrid model.Our construction permits the secure computation of multiple RAM programs on a persistent database, although we do not allow the inputs to be chosen adaptively by an attacker (a weakness that is present in, and inherited from garbled circuits). Furthermore, we inherit many of the optimizations found in garbled circuits, for example, garbled input in our construction can be compressed from $|x| \cdot \mathsf{poly}(\kappa)$ to $|x| + \mathsf{poly}(\kappa)$ via [AIKW13].

2 Our Techniques

The starting point for our work is the garbled RAM construction of Lu and Ostrovsky [LO13b]. Though the security of this construction is based on a strong and somewhat complex "circular" assumptions involving Yao garbled circuits and PRFs, it serves as a good starting point in explaining the ideas behind our construction based on one-way functions.

Starting point — Lu-Ostrovsky construction. The Lu-Ostrovsky construction views the program P, to be garbled, as a sequence of T CPU steps. Each of these CPU steps is represented as a circuit. Each CPU step reads or writes one bit of the RAM, which stores some dataset D. In order to keep this intuitive description simple, we will restrict ourselves to RAM programs that only read from memory and also to the weaker security requirement of unprotected memory access (UMA) in which we do not try to hide the database being stored or the memory locations being accessed (only the program and input is hidden). As noted in previous works [LO13b, GHL+14] this weaker security guarantee can be amplified to full security by using oblivious RAM. Garbling the RAM program itself involves just garbling the T CPU step circuits, using a circuit garbling scheme, e.g. Yao [Yao82]. The novelty is the added functionality of the ability to read bits from arbitrary memory locations without knowing them in advance or using interaction.

For each memory location i, containing value b_i the value $F_s(i, b_i)$ is stored in the "garbled" memory, where s is a secret PRF key. Let's consider that a CPU step that wants to read memory location i that needs to be fed into the next CPU step. Note that both these circuits will be independently garbled using Yao's garbled circuit technique. Let label^0 and label^1 be the garbled input wire labels corresponding to the wire for the read bit of the second circuit. In order to enable evaluation of the second garbled circuit, we need to be able to reveal *exactly one* of these two labels, corresponding to b_i, to the evaluator. Note that the first garbled circuit needs to do this without knowing i and b_i at the time of garbling. The idea for enabling the read is for the first garbled circuit to produce a translation gadget: the first garbled circuit outputs encryptions of labels label^0 and label^1 under keys $F_s(i, 0)$ and $F_s(i, 1)$ respectively. Since the evaluator holding the garbled memory only has one of the two values $F_s(i, 0)$ or $F_s(i, 1)$ at his disposal, he can only obtain either label^0 or label^1. This enables the evaluator to feed the proper bit into the next CPU step and continue the evaluation. Again we note here that since the location i that needs to be read is generated dynamically at runtime, we need the CPU step to be able to generate PRF values $F_s(i, 0)$ and $F_s(i, 1)$ dynamically, and in order to enable this computation we need to hardwire the secret key s in each of these CPU step circuits that are being garbled.

Circularity Assumption. As noted in [GHRW14a], in arguing security of the above scheme we need to rely of the security of Yao's garbled circuit, which in turn needs that only one label for each of its input wires is given out. Finally this needs to rely on the pseudorandomness property of the outputs of F. However the problem is that the key of the PRF s, is embedded inside the garbled circuits. Because of this circularity, the security of this construction requires a somewhat complicated assumption.

Gentry et al. [GHL+14] propose two solutions to this problem. At a high level, the common idea in the two solutions is to not embed the same PRF key in each CPU step. Instead they consider a sequence of keys of decreasing power that are hardcoded in the CPU steps. Abstractly this sequence of hardcoded keys is such that, a key hardcoded in a circuit can be used to decrypt the input labels for future garbled circuits but

not the current and past ones, breaking the circularity. Unfortunately, these solutions either require stronger assumptions, namely IBE [BF01], or more overhead that do not meet our poly-logarithmic efficiency goals.

There appears to be a fundamental barrier in all previous schemes: to read a value from a memory location, we need to have a key hardcoded in the garbled circuits that produces two values for any memory location, one of which is in the garbled memory. At the same time we need to claim that the value not in memory is indistinguishable for random in order to use Yao's security for the next garbled circuit. Given that we do not want to assume circularity the need to successively weaker keys seems unavoidable. The main technical question is how can we overcome this dependency without necessarily having to weaken keys?

Our Idea. Our main idea is to replace the process of key revocation, with the idea of *key evolution*. Data garbling in previous constructions was done using one master key that encrypted everything, and weakenings of which were hardcoded inside different garbled circuits, in an attempt to break circularity. Our strategy will be to have multiple keys, more explicitly a key tree, and anytime a key is used in a way that might effect the security of the later garbled circuits, then we immediately discard it and replace it with a fresh new key. Of course as a key is removed from the system we need to ensure that no value is encrypted under that key, as those values would become unusable once this key is discarded. Therefore if any value is encrypted under the key being discarder then that value must be first recovered and encrypted under the new key. In order to give a better intuition of this key evolution process we will start by describing how we garble the memory and then explain how this key evolution helps break circularity.

Here is how our memory is garbled: we sample a tree of fresh random keys $s^{i,j} \in \{0,1\}^\kappa$ where $i \in \{0, \dots d-1\}$ and $j \in \{0, \dots 2^i - 1\}$ where $d = \log(m/\kappa)$ and m is the size of the database D. The garbled memory itself consists of encryptions of the data under the leaf keys from the key tree and the encryption of each key in the key tree under its parent key. We refer the reader to look at Figure 1 for a graphical representation of the same. Note that given the root key $s^{0,0}$, starting from the root, one can navigate the tree and reach any leaf key $s^{d-1,j}$ with $d-1$ decryptions, which can be then used to read the desired bit from memory. On the other hand, withholding the root key, renders the entire tree hidden.

As already pointed, having access to the root key enables reading any bit from memory, and this process involves reading a sequence of $d-1$ keys from memory. Our idea of key evolution is that in the process of reading a bit from memory we will expunge all the keys along the path from the root to the leaf and replace them with freshly sampled keys. Note that since each key only encrypts two other keys or 2κ bits of memory, it is easy to read those values and output additionally an encryption of these values under the updated key. In other words, as a circuit navigates the tree, it will update all the nodes visited along the path using fresh keys. The additional subtlety here is that, whenever we replace a node with fresh key, then both its children nodes need to be re-encrypted under the fresh key. We believe that this is a novel approach on tree-based constructions in cryptography, e.g. it differs drastically from statistical ORAM [Ajt10, DMN11, SvDS+13, CP13], Merkle trees [Mer87], GGM PRF [GGM84] and broadcast encryption [FN93] and we expect this to have other applications in cryptography.

Now we explain how this key evolution process solves the circularity problem. We note that at any point in time, any key that is ever used to read any other key or data from memory has already been expunged. This allows us to claim the invariant that at point of time the evolved key tree only consists of pristine keys, that have never been used to read anything from memory. This gives the guarantee that future time steps have essentially nothing to do with the keys that were actively used in previous time steps.[1]

Finally note that the key evolution only changes the keys in the system but the size of the garbled memory itself does not change. Hence our solution only requires $\mathsf{poly}(\kappa)$ overhead to store the garbed tree and $\mathsf{poly}(\kappa, \log m)$ additional overhead in the running time of the CPU circuit, required to navigate the tree achieving the desired efficiency goals.

[1] Since this invariant is maintained across all timesteps, we achieve an interesting property that our solution does not need any mechanism to remember when a location was last accessed simplifying our construction significantly with respect to previous ones.

We note that in the proof various additional subtleties arise. In the life time of a key it might be read and re-encrypted multiple times, depending on the execution path of the program. The invariant above only claims that the key itself was not used to read anything from memory or in other words PRF values were not computed using this key. In the proof before we can rely on the security of PRF for this key we need to replace all the encryptions of this key with encryptions of random strings. We prove that this can indeed be done as all those encryptions are under keys that have already been expunged and so on.

Writing. Unlike previous GRAM schemes, where writing was more involved, our construction achieves writes in a very simple manner. Recall that reading in our scheme already involved re-encrypting the read data under its new parent. Writing just involves encrypting the value to be written instead of doing the re-encryption.

Decoupling of data garbling and program garbling. A very nice feature of our construction, that enables for various novel applications, is that the only connection between the garbled memory and the garbled program is in terms of the root key which can be fed into the garbled program rather than being hardcoded in it. This means that we can garble the program independent of the data.

2.1 Roadmap

We now lay out a roadmap for the remainder of the paper. In Section 3, we give necessary background and definitions for the RAM model, garbled circuits, and garbled RAM. In Section 4 we give the main construction of our result, and prove the security in Section 5. We survey other related work in Appendix A. We review oblivious RAM in Appendix B. In Appendix C we give a warmup construction of fully secure single-program GRAM from UMA-secure single-program GRAM (such a proof was previously given in [GHL+14], but given our slightly stronger definitions, we re-prove the result under these stronger conditions). In Appendix D we show how to obtain fully secure multi-program GRAM from UMA-secure multi-program GRAM.

3 Background

In this section we fix notation for RAM computation and provide formal definitions for Garbled Circuits and Garbled RAM Programs. Parts of this section have been taken verbatim from [GHL+14].

3.1 RAM Model

Notation for RAM Computation. We start by fixing the notation for describing standard RAM computation. For a program P with memory of size m we denote the initial contents of the memory data by $D \in \{0,1\}^m$. Additionally, the program gets a "short" input $x \in \{0,1\}^n$, which we alternatively think of as the initial state of the program. We use the notation $P^D(x)$ to denote the execution of program P with initial memory contents D and input x. The program can P read from and and write to various locations in memory D throughout its execution.[2]

We will also consider the case where several different programs are executed sequentially and the memory persists between executions. We denote this process as $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$ to indicate that first $P_1^D(x_1)$ is executed, resulting in some memory contents D_1 and output y_1, then $P_2^{D_1}(x_2)$ is executed resulting in some memory contents D_2 and output y_2 etc. As an example, imagine that D is a huge database and the programs P_i are database queries that can read and possibly write to the database and are parameterized by some values x_i.

CPU-Step Circuit. Consider a RAM program who execution involves at most T CPU steps. We represent a RAM program P via a sequence of T small *CPU-Step Circuit* where each of them executes a single CPU

[2] In general, the distinction between what to include in the program P, the memory data D and the short input x can be somewhat arbitrary. However as motivated by our applications we will typically be interested in a setting where that data D is large while the size of the program $|P|$ and input length n is small.

step. In this work we will denote one CPU step by:

$$C^P_{\mathsf{CPU}}(\mathsf{state}, z^{\mathsf{read}}) = (\mathsf{state}', L, z^{\mathsf{write}}).$$

This circuit takes as input the current CPU state state and a block z^{read}. Looking ahead this block will be read from the memory location that was requested for a memory location requested for in the previous CPU step. The CPU step outputs an updated state state', the next location to read $L \in [m]$, and a block z^{write} to write into the location that was previously read. The sequence of locations and read/write values collectively form what is known as the *access pattern*, namely $\mathsf{MemAccess} = \{(L^\tau, z^{\mathsf{read},\tau}, z^{\mathsf{write},\tau}) : \tau = 1, \ldots, t\}$.

Note that in the description above without loss of generality we have made some simplifying assumptions. First, we assume that the output z^{write} is written into the same location z^{read} was read from. Note that this is sufficient to both read from and write to arbitrary memory locations. Secondly we note that we assume that each CPU-step circuit always reads from and write some location in memory. This is easy to implement via a dummy read and write step. Finally, we assume that the instructions of the program itself is hardwired into the CPU-step circuits, and the program can first load itself into memory before execution. In cases where the size of the program vastly differs from its running time, one can suitably partition the program into two pieces.

Representing RAM computation by CPU-Step Circuits. The computation $P^D(x)$ starts with the initial state set as $\mathsf{state}_0 = x$ and initial read location $L_0 = 0$ as a dummy read operation. In each step $\tau \in \{0, \ldots T-1\}$, the computation proceeds by first reading memory location L^τ, that is by setting $b^{\mathsf{read},\tau} := D[L^\tau]$ if $\tau \in \{1, \ldots T-1\}$ and as 0 if $\tau = 0$. Next it executes the CPU-Step Circuit $C^P_{\mathsf{CPU}}(\mathsf{state}^\tau, b^{\mathsf{read},\tau}) = (\mathsf{state}^{\tau+1}, L^{\tau+1}, b^{\mathsf{write},\tau+1})$. Finally we write to the location L^τ by setting $D[L^\tau] := b^{\mathsf{write},\tau+1}$. If $\tau = T-1$ then we set state to be the output of the program P and ignore the value $L^{\tau+1}$. Note here that we have without loss of generality assumed that in one step the CPU-Step the same location in memory is read from and written to. This has been done for the sake of simplifying exposition.

3.2 Garbled Circuits

Garbled circuits was first constructed by Yao [Yao82] (see Lindell and Pinkas [LP09] and Bellare et al. [BHR12] for a detailed proof and further discussion). A circuit garbling scheme is a tuple of PPT algorithms $(\mathsf{GCircuit}, \mathsf{Eval})$. Very roughly $\mathsf{GCircuit}$ is the circuit garbling procedure and Eval the corresponding evaluation procedure. Looking ahead, each individual wire w of the circuit will be associated with two labels, namely $\mathsf{lab}^w_0, \mathsf{lab}^w_1$. Finally, since one can apply a generic transformation (see, e.g. [AIK10]) to blind the output, we allow output wires to also have arbitrary labels associated with them. Indeed, we can classify the output values into two categories — *plain outputs* and *labeled outputs*. The difference in the two categories stems from how they will be treated when garbled during garbling and evaluation. The plain output values do not require labels provided for them and evaluate to cleartext values. On the other hand labeled output values will require that additional output labels be provided to $\mathsf{GCircuit}$ at the time of garbling, and Eval will only return these output labels and not the underlying cleartext. We also define a well-formedness test for labels which we call Test.

- $\left(\tilde{C}, \{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{inp}(C), b \in \{0,1\}}\right) \leftarrow \mathsf{GCircuit}\left(1^\kappa, C, \{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{out}(C), b \in \{0,1\}}\right)$: $\mathsf{GCircuit}$ takes as input a security parameter κ, a circuit C, and a set of labels lab^w_b for all the output wires $w \in \mathsf{out}(C)$ and $b \in \{0,1\}$. We denote the set of input and output wires by $\mathsf{inp}(C)$ and $\mathsf{out}(C)$ respectively. This procedure outputs a *garbled circuit* $\tilde{C}$ and a set of labels lab^w_b for each input wire $w \in \mathsf{inp}(C)$ and $b \in \{0,1\}$.
- $0/1 \leftarrow \mathsf{Test}(\mathsf{lab})$ tests whether a label is well-formed.

- $\left(\{(w, o_w)\}_{w \in \mathsf{out}(C)}\right) = \mathsf{Eval}(\tilde{C}, \{(w, \mathsf{lab}^w_{x_w})\}_{w \in \mathsf{inp}(C)})$: Given a garbled circuit $\tilde{C}$ and a sequence of input labels $\{(w, \mathsf{lab}^w_{x_w})\}_{w \in \mathsf{inp}(C)}$, Eval outputs a sequence of outputs $\{(w, o)\}_{w \in \mathsf{out}(C)}$, where each o_w is either y_w or $\mathsf{lab}^w_{y_w}$ depending on whether the output is being given in the clear or as a label only.

Correctness. For correctness, we require that for any circuit C and input $x \in \{0,1\}^n$ (here n is the input length to C) we have that that:

$$\Pr\left[\left(\{(w, y_w, \mathsf{lab}^w_{y_w})\}_{w \in \mathsf{out}(C)}\right) = \mathsf{Eval}(\tilde{C}, \{(w, \mathsf{lab}^w_{x_w})\}_{w \in \mathsf{inp}(C)})\right] = 1$$

where $\left(\tilde{C}, \{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{inp}(C), b \in \{0,1\}}\right) \leftarrow \mathsf{GCircuit}\left(1^\kappa, C, \{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{out}(C), b \in \{0,1\}}\right)$ and $y = C(x)$. We require that the testing algorithm works correctly: $\mathsf{Test}(\mathsf{lab}) = 1$ if $(\cdot, \cdot, \mathsf{lab}) \in \{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{inp}(C), b \in \{0,1\}}$, where $\left(\tilde{C}, \{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{inp}(C), b \in \{0,1\}}\right) \leftarrow \mathsf{GCircuit}\left(1^\kappa, C, \{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{out}(C), b \in \{0,1\}}\right)$, and $\Pr[\mathsf{Test}(r) = 1]$ is negligible when r is uniform. This can be implemented, for example, by enforcing all labels have a $O(k)$-size padding of 0 bits.

Security. For security, we require that there is a PPT simulator Sim such that for any C, x, and labels $\left(\{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{out}(C), b \in \{0,1\}}\right)$, we have that:

$$\left(\tilde{C}, \{(w, \mathsf{lab}^w_{x_w})\}_{w \in \mathsf{inp}(C)}\right) \overset{\text{comp}}{\approx} \mathsf{Sim}\left(1^\kappa, \{(w, b, \mathsf{lab}^w_{y_w})\}_{w \in \mathsf{out}(C)}\right)$$

where $\left(\tilde{C}, \{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{inp}(C), b \in \{0,1\}}\right) \leftarrow \mathsf{GCircuit}\left(1^\kappa, C, \{(w \mathsf{lab}^w_b)\}_{w \in \mathsf{out}(C), b \in \{0,1\}}\right)$ and $y = C(x)$.

3.3 Garbled RAM

Next we consider an extension of garbled circuits to the setting of RAM programs. In this setting the memory data D is garbled once and then many different garbled programs can be executed sequentially with the memory changes persisting from one execution to the next. We will start by presenting our definitions for the case when only one program is garbled and then present the definitions for the case when multiple programs are garbled in the Appendix. Another simplifying assumption we make is that in our definition here, we focus on a weaker variant (that also appeared in [GHL+14]) known as Unprotected Memory Access (UMA) , and we define full security in the Appendix and show how UMA-secure Garbled RAM can be compiled with Oblivious RAM to achieve full security.

Syntax. A *UMA-secure single-program garbled RAM* scheme consists of four procedures: $(\mathsf{GData}, \mathsf{GProg}, \mathsf{GInput}, \mathsf{GEval})$ with the following syntax:

- $(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D)$: Given a security parameter 1^κ and memory $D \in \{0,1\}^m$ as input GData outputs the garbled memory $\tilde{D}$.
- $(\tilde{P}, s^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log m}, 1^t, P)$: Takes the description of a RAM program P with memory-size m as input. It then outputs a garbled program $\tilde{P}$ and an input-garbling-key s^{in}.
- $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x, s^{in}, s)$: Takes as input $x \in \{0,1\}^n$ and and an input-garbling-key s^{in}, a garbled "tree root" key s and outputs a garbled-input $\tilde{x}$.
- $y = \mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x})$: Takes a garbled program $\tilde{P}$, garbled input $\tilde{x}$ and garbled memory data $\tilde{D}$ and output a value y. We model GEval itself as a RAM program that can read and write to arbitrary locations of its memory initially containing $\tilde{D}$.

Efficiency. We require the run-time of GProg and GEval to be $t \cdot \mathsf{poly}(\kappa) \cdot \mathsf{polylog}(m)$, which also serves as the bound on the size of the garbled program $\tilde{P}$. Moreover, we require that the run-time of GData should be $m \cdot \mathsf{polylog}(m) \cdot \mathsf{poly}(\kappa)$, which also serves as an upper bound on the size of $\tilde{D}$. Finally the running time of GInput is required to be $n \cdot \mathsf{poly}(\kappa)$.

Correctness. For correctness, we require that for any program P, initial memory data $D \in \{0,1\}^m$ and input x we have that:

$$\Pr[\mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x}) = P^D(x)] = 1$$

where $(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D)$, $(\tilde{P}, s^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log m}, 1^t, P)$, $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x, s^{in}, s)$.

Security with Unprotected Memory Access (UMA). For security, we require that there exists a PPT simulator Sim such that for any program P, initial memory data $D \in \{0,1\}^m$ and input x, which induces access pattern $\mathsf{MemAccess}$ we have that:

$$(\tilde{D}, \tilde{P}, \tilde{x}) \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\kappa, 1^m, 1^t, y, D, \mathsf{MemAccess})$$

where
$(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D)$, $(\tilde{P}, s^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log m}, 1^t, P)$ and $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x, s^{in}, s)$, and $y = P^D(x)$.

4 The Construction

In this section we describe our construction for garbled RAM formally, namely the procedures (GData, GProg, GInput, GEval). We use the notation $[n]$ to denote the set $\{0, \ldots, n-1\}$. Throughout the construction, we let $F : \{0,1\}^* \to \{0,1\}^\kappa$ be a PRF with seed length κ. For any string x we reserve the use of subscript to denote its bit locations. For example for a string x, we use x_i to denote the i^{th} bit of x where $i \in [|x|]$ with the 0^{th} bit being the highest order bit.

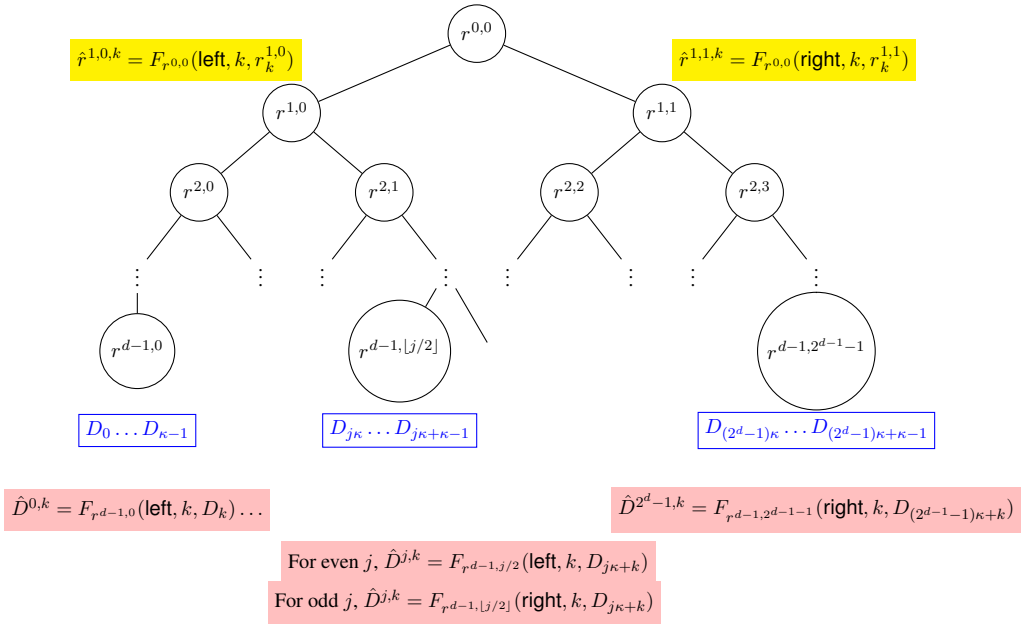

Figure 1: Visualization of Memory Garbling.

4.1 Data Garbling: $(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D)$

We start by providing an informal description of the data garbling procedure. The formal description is provided in Figure 2. To explain the memory garbling procedure better we provide its graphical representation in Figure 1. In Figure 1 the memory D of size $m = m' \cdot \kappa$, where $m' = 2^d$, is represented in the blue rectangular boxes. In order to garble D, we sample PRF keys $r^{i,j} \leftarrow \{0,1\}^\kappa$, where $i \in [d]$ and $j \in [2^i]$.

In the figure these keys are shown on in the black circles. Note that at the lowest key the tree consists of 2^{d-1} keys. These keys are what are used to encrypt the data. In particular for any even $j \in [2^d]$, a total of 2κ bits of D, namely $D_{j\kappa} \dots D_{j\kappa+2\kappa-1}$ are encrypted using the key $r^{d-1,j/2}$. The first κ of these are encrypted with a tag left and the next κ ones are encrypted with the tag right. For example in Figure 1 we show the encryption of first κ bits of D, the last κ bits of D and $j\kappa$ to $j\kappa + \kappa - 1$ bits of D, under keys $r^{d-1,0}$, $r^{d-1,2^{d-1}-1}$ and $r^{d-1,\lfloor j/2 \rfloor}$ respectively. These encryptions are colored in pink.

The $\mathsf{GData}(1^\kappa, D)$ procedure proceeds as follows.

1. Sample $\{r^{i,j}\}_{i\in[d],j\in[2^i]} \leftarrow \{0,1\}^\kappa$.
2. Without loss of generality we assume that $m = 2^d \cdot \kappa$ where d is a positive integer and set $m' = 2^d$. For $j \in [m']$ and $k \in [\kappa]$, if $j \mod 2 = 0$ set $\hat{D}^{j,k} = F_{r^{d-1,j/2}}(\mathsf{left}, k, D_{j\kappa+k})$ else set $\hat{D}^{j,k} = F_{r^{d-1,\lfloor j/2 \rfloor}}(\mathsf{right}, k, D_{j\kappa+k})$, where $D_{j\kappa+k}$ denotes the $j\kappa + k^{th}$ bit of D.
3. $\forall i \in \{1, \dots d-1\}, j \in [2^i], k \in [\kappa]$, if $j \mod 2 = 0$ then compute $\hat{r}^{i,j,k} = F_{r^{i-1,\lfloor j/2 \rfloor}}(\mathsf{left}, k, r^{i,j}_k)$ else compute $\hat{r}^{i,j,k} = F_{r^{i-1,\lfloor j/2 \rfloor}}(\mathsf{right}, k, r^{i,j}_k)$, where $r^{i,j}_k$ denotes the k^{th} bit of $r^{i,j}$.
4. Output $\tilde{D} = (\{\hat{r}^{i,j,k}\}_{i\in[d]\setminus\{0\},j\in[2^i],k\in[\kappa]}, \{\hat{D}^{j,k}\}_{j\in[2^d],k\in[\kappa]})$ and $s = r^{0,0}$.

Figure 2: Formal description of GData.

We also generate encryptions of each key in the tree under its parent. Specifically, each bit of the key $r^{i,j}$ is encrypted under the key $r^{i-1,\lfloor j/2 \rfloor}$. For example in Figure 1 we provide encryptions of $r^{1,0}$ and $r^{1,1}$ under $r^{0,0}$ in yellow color. The garbled memory provided consists of the generated encryptions of the provided keys and the memory.

4.2 Program Garbling: $(\tilde{P}, s^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log m}, 1^t, P)$

We start by defining three sub-circuits and some notation that will be needed in describing the garbling itself.

Our Sub-Circuits. We use the notation $C^{\mathsf{type}}[\mathsf{param}]$ to describe a circuit C^{type} that has hardwired parameters param, where $\mathsf{type} \in \{\mathsf{nav}, \mathsf{step}, \mathsf{final}\}$ the three types of circuits we will define. These circuits will be referred to as the *navigation* circuit, the *step* circuit and the *final* circuit respectively.

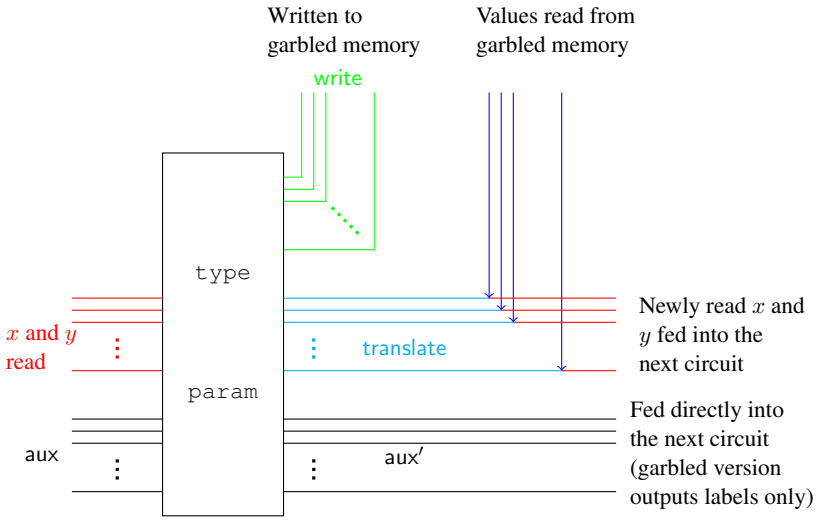

Figure 3: Visualization of $C^{\mathsf{type}}[\mathsf{param}]$.

Input-output behavior of these circuits. Each one of these three circuits takes as input (x, y, aux), where $x, y \in \{0,1\}^\kappa$ and $\mathsf{aux} = (\mathsf{state}, L)$ where $\mathsf{state} \in \{0,1\}^*$ and $L \in \{0,1\}^d$. Looking ahead, x, y denote the 2κ bits just read from garbled memory, state represents the input state of the CPU computation including location L that describes the block of data memory we are currently interested in reading.

1. **Navigation Circuit** $C^{\mathsf{nav}}[r, r', i, \overline{\mathsf{lab}}]$: This circuit has hardwired in it PRF keys r and r', value $i \in [d]$ and labels $\overline{\mathsf{lab}} = \{\mathsf{lab}^{\mathsf{left},k,b}, \mathsf{lab}^{\mathsf{right},k,b}\}_{k\in[k],b\in\{0,1\}}$. On input $x, y \in \{0,1\}^\kappa$, $\mathsf{aux} = (\mathsf{state}, L) \in \{0,1\}^* \times \{0,1\}^d$ the circuit output is generated as follows.

   (a) If $L_i = 0$ then set $key = x, newL = r', newR = y$ else set $key = y, newL = x, newR = r'$ and set $\mathsf{write} := (L, \{F_r(\mathsf{left}, k, newL_k), F_r(\mathsf{right}, k, newR_k)\}_{k\in[\kappa]})$,

$$\mathsf{translate} := \left\{ \begin{matrix} F_{key}(\mathsf{left}, k, 0) \oplus \mathsf{lab}^{\mathsf{left},k,0}, & F_{key}(\mathsf{right}, k, 0) \oplus \mathsf{lab}^{\mathsf{right},k,0}, \\ F_{key}(\mathsf{left}, k, 1) \oplus \mathsf{lab}^{\mathsf{left},k,1}, & F_{key}(\mathsf{right}, k, 1) \oplus \mathsf{lab}^{\mathsf{right},k,1}. \end{matrix} \right\}_{k\in[\kappa]}$$

   (b) Randomly permute the rows of $\mathsf{translate}$ and output $(\mathsf{write}, \mathsf{translate}, \mathsf{aux})$.

2. **Step Circuit** $C^{\mathsf{step}}[r, s, \overline{\mathsf{lab}}]$: This circuit has hardwired in it PRF keys r and s and labels $\overline{\mathsf{lab}} = \{\mathsf{lab}^{\mathsf{left},k,b}, \mathsf{lab}^{\mathsf{right},k,b}\}_{k\in[k],b\in\{0,1\}}$. On input $x, y \in \{0,1\}^\kappa$, $\mathsf{aux} = (\mathsf{state}, L) \in \{0,1\}^* \times \{0,1\}^d$ the circuit output is generated as follows.

   (a) Set $newL = x, newR = y$. If $L_{d-1} = 0$ then set $read = x$ else $read = y$. Compute $(\mathsf{state}', L', z) := C^P_{\mathsf{CPU}}(\mathsf{state}, read)$. If $L_{d-1} = 0$ then overwrite $newL = z$ else overwrite $newR = z$.

   (b) Compute $\mathsf{write} := (L', \{F_r(\mathsf{left}, k, newL_k), F_r(\mathsf{right}, k, newR_k)\}_{k\in[\kappa]})$,

$$\mathsf{translate} := \left\{ \begin{matrix} F_s(\mathsf{left}, k, 0) \oplus \mathsf{lab}^{\mathsf{left},k,0}, & F_s(\mathsf{right}, k, 0) \oplus \mathsf{lab}^{\mathsf{right},k,0}, \\ F_s(\mathsf{left}, k, 1) \oplus \mathsf{lab}^{\mathsf{left},k,1}, & F_s(\mathsf{right}, k, 1) \oplus \mathsf{lab}^{\mathsf{right},k,1}. \end{matrix} \right\}_{k\in[\kappa]}$$

   (c) Randomly permute the rows of $\mathsf{translate}$ and output $(\mathsf{write}, \mathsf{translate}, \mathsf{aux}')$ where $\mathsf{aux}' := (\mathsf{state}', L')$.

3. **Final Circuit** $C^{\mathsf{final}}[r]$: This circuit is similar to the circuit $C^{\mathsf{step}}[r, s]$ but with part of its functionality trimmed. This circuit has hardwired in it PRF keys r. On input $x, y \in \{0,1\}^\kappa$, $\mathsf{aux} = (\mathsf{state}, L) \in \{0,1\}^* \times \{0,1\}^d$ the circuit output is generated as follows.

   (a) Set $newL = x, newR = y$. If $L_{d-1} = 0$ then set $read = x$ else $read = y$. Compute $(\mathsf{state}', L', z) := C^P_{\mathsf{CPU}}(\mathsf{state}, read)$. If $L_{d-1} = 0$ then overwrite $newL = z$ else overwrite $newR = z$. Compute $\mathsf{write} := (L', \{F_r(\mathsf{left}, k, newL_k), F_r(\mathsf{right}, k, newR_k)\}_{k\in[\kappa]})$.

   (b) Output $(\mathsf{write}, \mathsf{aux}')$ where $\mathsf{aux}' := (\mathsf{state}', L')$.

Figure 4: Formal description of subcircuits for GProg.

The output of C^{nav} and C^{step} consists of $(\mathsf{write}, \mathsf{translate}, \mathsf{aux}')$. Roughly the output write will consist of information that will enable writing something into memory, $\mathsf{translate}$ will consist of values that enable reading from memory and finally aux' just describes the output CPU-state including location L that describes the block of data memory we are currently interested in reading. The output of the final circuit C^{final} is just

$(\mathsf{write}, \mathsf{aux})$. Note that the final circuit is essentially same as the step circuit except the functionality that generates $\mathsf{translate}$ has been trimmed. This is depicted in Figure 3.

Communication between different circuits. Looking ahead, our garbling of a RAM program will consist of garbling of multiple copies of these three circuits instantiated with different parameters hardwired into them. Furthermore we will need this garbling to be such that these garbled circuits can "talk to each other." This communication will be enabled in two ways: 1) We directly pass the output of one circuit into the input of another. This can be achieved by having the first garbled circuit produce as output the labels needed for the inputs of the next garbled circuit. 2) The garbled memory is involved, where in particular, one garbled circuit will output a translation table which will encode pairs of labels for input wires of the second circuit it wants to communicate with. Given this translation table depending on bits stored in the garbled memory the evaluator will be able to obtain exactly 1-out-of-2 of the labels. This will be tantamount to reading from the underlying memory. The translation information corresponds precisely to the $\mathsf{translate}$ output of the navigation and the step circuits. More specifically, let $\overline{\mathsf{lab}} = \{\mathsf{lab}^{\mathsf{left},k,b}, \mathsf{lab}^{\mathsf{right},k,b}\}_{k\in[\kappa],b\in\{0,1\}}$ be the labels for input wires of a circuit with which the circuit at hand is trying to communicate. Since the translation table $\mathsf{translate}$ generated by the circuit at hand needs to depend on the labels $\overline{\mathsf{lab}}$, we will need to hard-code these labels in the circuit being garbled.

We detail the subcircuits in Figure 4. The process of garbling multiple instances of these circuits and their process of communication will be explained later.

The actual garbling. We now provide an informal description of our RAM garbling procedure. A formal description of GProg is provided in Figure 6.

As mentioned earlier, garbling a RAM program will involve garbling multiple instances of circuits described earlier. Specifically if the running time of the program P being garbled is t and the size of the database is $m = m' \cdot \kappa$, then the garbled RAM program will consist of $t \cdot \log m'$ garbled circuits which we can intuitively group into $\log m'$ garbled circuits for each time step of the program. In particular for each time step we will consider a sequence of $\log m' - 1$ instances of C^{nav} followed by one instance of C^{step}. For the last time step we just replace the last C^{step} circuit with C^{final}. Recall that the only difference between C^{step} and C^{final} is that unlike C^{step}, C^{final} does not output any translate information $\mathsf{translate}$.

Now that we have explained the overall structure, we will next describe the role of each of these circuits. It is helpful to keep in mind how the garbled memory is constructed using a tree of keys (cf. Fig 2). In short, the *plain* version simply consists of a tree of randomly sampled PRF keys for each non-leaf node, and a κ bit data block from database D for each leaf. The *encrypted* version of this plain version, which constitutes the actual garbled memory, consists of an encryption of each PRF key under its parent, with the root removed. More specifically, a non-root PRF key (or a database block) r is encrypted as $\{F_s(\mathsf{left}/\mathsf{right}, k, r_k)\}_{k\in[\kappa]}$, where $\mathsf{left}/\mathsf{right}$ indicates if the node is a left or right child, s is the PRF key of its parent, and r_k is the k^{th} bit of the PRF key (or the database block).

For each step $\tau \in [t]$ of the computation, the step circuit C^{step} requires a κ-bit block of data from memory at location L. However this is encrypted under a key from level $d-1$. This key is itself encrypted under another key from level $d-2$ and so on. Hence in order to read the desired memory location, we need to navigate in the memory from level 0 to level $d-1$ reading one key at a time and finally recovering the desired data block. This is exactly what will be achieved by a sequence of $\log m'$ circuits. As we will see next the last of these circuits also performs the computation corresponding to the time step and help kick-start the next computation step. This depicted informally in Figure 5.

More concretely the i^{th} circuit for $i \in [d]$ takes as input two sibling PRF keys and uses one among them to decrypt and obtain PRF keys corresponding to its two children, depending on what data block we actually want to read. For example, the 0^{th} circuit takes as input the PRF keys for the two children of the root. It decrypts the two children of one of these two nodes, depending on what data block from memory is to be read. All the other circuits behave analogously. The last circuit will have the root PRF key embedded in it

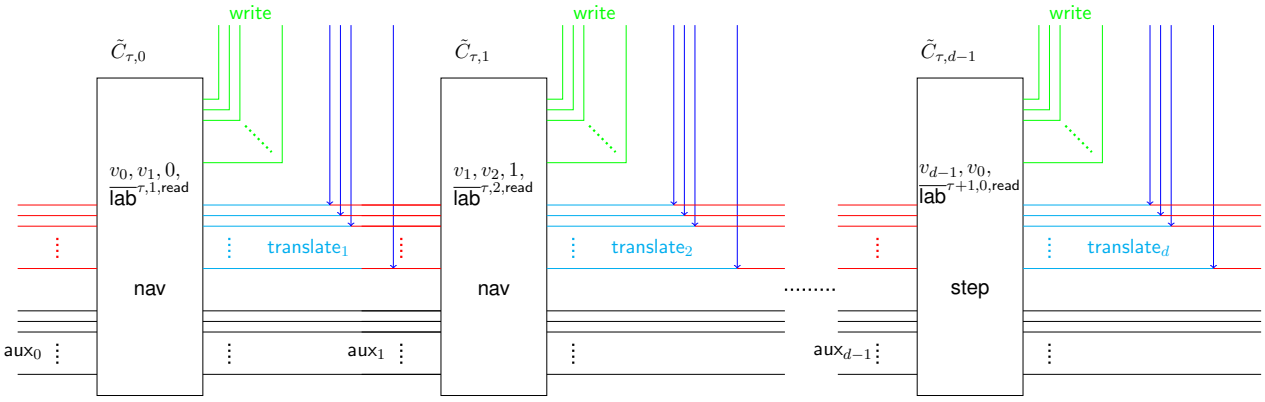

Figure 5: Visualization of one time step of the RAM Program.

and this would enable it to read the two children of the root needed for the next $\log m'$ sequence of garbled circuits needed for the next time step. Additionally for security, each step will also replace each key along the path that has been used to decrypt anything else with a fresh key. The reason for this is that for security we maintain the variant that any PRF key in the system is used at most once for reading anything. So any time a PRF key is used to decrypt its two children then then that PRF key is immediately updated to fresh key.More details on why this is needed will be elaborated on in the proof.

A bit more precisely, we consider the semantics of $C^{\mathsf{nav}}[r, r', i, \overline{\mathsf{lab}}](x, y, \mathsf{aux})$ which will output $(\mathsf{write}, \mathsf{translate}, \mathsf{aux})$. Here r' is a freshly chosen PRF key that will replace either the plain value of the one of the two PRF keys it gets as input, depending on the path we do end up taking, which in turn depends on the memory block we are interested in reading. The PRF key r is the PRF key that replaced the PRF key of the parent of the two nodes in consideration. Since the key of the parent has been updated we need to to re-encrypt the input PRF keys under r. At a high level, in reading some data location L we replace all the keys, along the path in the tree from the root to the leaf, with fresh PRF keys. For consistency this requires that the children of all these nodes corresponding to which keys have have updated be re-encrypted under these fresh keys. Interestingly this includes the siblings that haven't even been used. We stress that even though a priori we do not not know which values in key tree will be replaced with fresh values; we do know that the values they will be replaced with (they are freshly chosen) and these values are what are hardwired into the circuits.

Finally, we describe the other two circuits C^{step} and C^{final} in terms of how they differ from C^{nav} a bit more. These circuits can be considered as virtual navigate circuits at level $d-1$: the input consist of two leaf nodes which correspond to actual memory that can be read from, and it helps in enabling "wrap around." In other words a mechanism that enables reading the PRF keys stored in the two children of the root node. Also, unlike C^{nav} which replaces each key used to read other keys with a fresh key, this circuit executes the underlying CPU step and writes back the block b^{write}. Finally, in order to obtain the translation table, these circuits will be hardwired with the value of the root node. Note that C^{final} is same as C^{step} except that it does not need to provide any "wrap around."

Now that we have roughly described the overall structure and the role of each individual circuit in the garbling process we will next describe a bit more precisely how the garbled circuits communicate. This is actually very simple. Each circuit $\tilde{C}^{\tau,i}$ passes on its output aux directly as input to the circuit $\tilde{C}^{\tau,i+1}$ if $i < d-1$ or to the circuit $\tilde{C}^{i+1,0}$ otherwise. Similarly the circuit $\tilde{C}^{\tau,i}$ provides translation information that enables evaluator of these garbled circuits to read a bit from memory.

Toward capturing this, we use the following notation. We denote the set of *all* input labels of circuit $\tilde{C}^{\tau,i}$ by $\overline{\mathsf{lab}}^{\tau,i}$. Then, within this set we distinguish two kind of labels: namely $\overline{\mathsf{lab}}^{\tau,i} = [\overline{\mathsf{lab}}^{\tau,i,\mathsf{read}}, \overline{\mathsf{lab}}^{\tau,i,\mathsf{aux}}]$,

where $\overline{\mathsf{lab}}^{\tau,i,\mathsf{read}}$ denotes the input labels corresponding to the input values x and y — the values just *read* from memory and $\overline{\mathsf{lab}}^{\tau,i,\mathsf{aux}}$ denotes the input labels corresponding to the input aux. Generating the garbled circuit $\tilde{C}^{\tau,i+1}$ requires additionally the information $\overline{\mathsf{lab}}^{\tau,i+1}$ if $i < d-1$ and $\overline{\mathsf{lab}}^{\tau+1,0}$ otherwise.

We note that since the generation of $\tilde{C}^{\tau,i+1}$ depends on labels $\overline{\mathsf{lab}}^{\tau,i+1}$ or $\overline{\mathsf{lab}}^{\tau+1,0}$, therefore we need to gable these circuits in the opposite order, i.e. garbling the last circuit first. As a result, during garbling we will know ahead of time what the input labels for the next garbled circuit will be.

The $\mathsf{GProg}(1^\kappa, 1^{\log m}, 1^t, P)$ procedure proceeds as follows. All garbled sub-circuits will output write and translate in the clear, so we omit assigning wire labels to them when invoking GCircuit. Given a garbled circuit $\tilde{C}^{\tau,i}$, we parse its input labels as $\overline{\mathsf{lab}}^{\tau,i} = [\overline{\mathsf{lab}}^{\tau,i,\mathsf{read}}, \overline{\mathsf{lab}}^{\tau,i,\mathsf{aux}}]$ where read corresponds to the inputs x and y.

1. Sample $u_1 \dots u_t \leftarrow \{0,1\}^\kappa$.
2. For each $\tau \in \{t-1, \dots 0\}$ proceed as follows:
   (a) Sample $v_1, v_2 \dots v_{d-1} \leftarrow \{0,1\}^\kappa$ and set $v_0 = u_{\tau+1}$.
   (b) If $\tau = t-1$ then generate $(\tilde{C}^{t-1,d-1}, \overline{\mathsf{lab}}^{t-1,d-1}) \leftarrow \mathsf{GCircuit}\left(1^\kappa, C^{\mathsf{final}}[v_{d-1}]\right)$, otherwise $(\tilde{C}^{\tau,d-1}, \overline{\mathsf{lab}}^{\tau,i}) \leftarrow \mathsf{GCircuit}\left(1^\kappa, C^{\mathsf{step}}\left[v_{d-1}, v_0, \overline{\mathsf{lab}}^{\tau+1,0,\mathsf{read}}\right], \overline{\mathsf{lab}}^{\tau+1,0,\mathsf{aux}}\right)$.
   (c) For $i \in \{d-2, \dots, 0\}$.
      i. Compute $(\tilde{C}^{\tau,i}, \overline{\mathsf{lab}}^{\tau,i}) \leftarrow \mathsf{GCircuit}\left(1^\kappa, C^{\mathsf{nav}}\left[v_i, v_{i+1}, i, \overline{\mathsf{lab}}^{\tau,i+1,\mathsf{read}}\right], \overline{\mathsf{lab}}^{\tau,i+1,\mathsf{aux}}\right)$.
3. Output $\tilde{P} = \{\tilde{C}^{\tau,i}\}_{\tau\in[t],i\in[d]}$, $s^{in} = \left(\overline{\mathsf{lab}}^{0,0,\mathsf{aux}}, \overline{\mathsf{lab}}^{0,0,\mathsf{read}}\right)$. We make note of the final root key u_t.

Figure 6: Formal description of GProg.

4.3 Input Garbling: $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x, s^{in}, s)$

Informally, the GInput algorithm uses x and s as selection bits for the labels provided by s^{in} and outputs $\tilde{x}$, which is just the selected labels. A formal description of GProg is provided in Figure 7.

The algorithm $\mathsf{GInput}(1^\kappa, x, s^{in}, s)$ proceeds as follows. Here $s^{in} = (\overline{\mathsf{lab}}^{\mathsf{aux}}, \overline{\mathsf{lab}}^{\mathsf{read}})$.

1. Parse $\overline{\mathsf{lab}}^{\mathsf{read}}$ as $\{\mathsf{lab}^{\mathsf{left},k,b}, \mathsf{lab}^{\mathsf{right},k,b}\} k \in [\kappa], b \in \{0,1\}$ and compute

$$\mathsf{translate} := \left\{ \begin{matrix} F_s(\mathsf{left},k,0) \oplus \mathsf{lab}^{\mathsf{left},k,0}, & F_s(\mathsf{right},k,0) \oplus \mathsf{lab}^{\mathsf{right},k,0}, \\ F_s(\mathsf{left},k,1) \oplus \mathsf{lab}^{\mathsf{left},k,1}, & F_s(\mathsf{right},k,1) \oplus \mathsf{lab}^{\mathsf{right},k,1}. \end{matrix} \right\}_{k\in[\kappa]}$$

2. Output $(\mathsf{translate}, \widehat{\mathsf{lab}}^{\mathsf{aux},\mathsf{state}=x,L=0})$, where $\widehat{\mathsf{lab}}^{\mathsf{aux},\mathsf{state}=x,L=0}$ selects the wire labels for $\mathsf{aux} = (\mathsf{state}, L)$ as $\widehat{\mathsf{lab}}^{\mathsf{aux},\mathsf{state}=x,L=0} = \{\mathsf{lab}^{\mathsf{aux},i,x_i}\}_{i\in\mathsf{inp}(\mathsf{state})}, \{\mathsf{lab}^{\mathsf{aux},i,0}\}_{i\in\mathsf{inp}(L)}$, where these are selected from the full set of labels $\overline{\mathsf{lab}}^{\mathsf{aux}} = \{\mathsf{lab}^{\mathsf{aux},i,b}\}_{i\in\mathsf{inp}(\mathsf{aux})=\mathsf{inp}(\mathsf{state},L),b\in\{0,1\}}$.

Figure 7: Formal description of GInput.

4.4 Garbled Evaluation: $y \leftarrow \mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x})$

The GEval procedure gets as input the garbled database $\tilde{D}$ is $(\{\hat{r}^{i,j,k}\}_{i\in[d-1]\setminus\{0\},j\in[2^i],k\in[\kappa]}, \{\hat{D}^{j,k}\}_{j\in[2^d],k\in[\kappa]})$, the garbled program $\tilde{P} = \{\tilde{C}^{\tau,i}\}_{\tau\in[t],i\in[d]}$ and the garbled input $\tilde{x} = (\mathsf{translate}^x, \widehat{\mathsf{lab}}^{\mathsf{aux},x})$. Intuitively the GEval is very simple. It proceeds by executing the sequence of garbled programs in the prescribed order. The labels needed to evaluate the first garbled circuit are provided as part of the garbled input and each evaluation of a garbled circuit reveals the labels needed for the next circuit. Evaluation of each garbled circuit also generates additional values for writing into memory and translation tables for reading values from memory. These are also carried out in the natural manner. Next we provide the formal description GEval. We will define a function $\mathsf{DeTranslate}(\mathsf{translate}, \{\hat{r}^{\mathsf{left},k}, \hat{r}^{\mathsf{right},k}\}_{k\in[\kappa]}, \tilde{C})$ that unblinds the labels one at a time. The function DeTranslate interprets

$$\mathsf{translate} = \left\{ \begin{matrix} \alpha^{\mathsf{left},k,0}, \alpha^{\mathsf{right},k,0} \\ \alpha^{\mathsf{left},k,1}, \alpha^{\mathsf{right},k,1} \end{matrix} \right\}$$

and outputs labels $\{\beta^{\mathsf{left},k}, \beta^{\mathsf{right},k}\}_{k\in[\kappa]}$ computed as follows: For each input wire x_i corresponding to x, for $b \in \{0,1\}$ if $\mathsf{Test}(\tilde{C}, (x_i, b, \alpha^{\mathsf{left},k,b} \oplus \hat{r}^{\mathsf{left},k})) = 1$ then set $\beta^{\mathsf{left},k} = \alpha^{\mathsf{left},k,b} \oplus \hat{r}^{\mathsf{left},k}$. Similarly, for each input wire y_i corresponding to y, for $b \in \{0,1\}$ if $\mathsf{Test}(\tilde{C}, (y_i, b, \alpha^{\mathsf{right},k,b} \oplus \hat{r}^{\mathsf{right},k})) = 1$ then set $\beta^{\mathsf{right},k} = \alpha^{\mathsf{right},k,b} \oplus \hat{r}^{\mathsf{right},k}$. A formal description of GProg is provided in Figure 8.

The algorithm $\mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x})$ proceeds as follows.

1. Initialize $L = 0$, $\mathsf{translate} := \mathsf{translate}^x$ and $\widehat{\mathsf{lab}} := \widehat{\mathsf{lab}}^{\mathsf{aux},x}$.
2. For $\tau \in [t]$,$i \in [d-1]$ do:
   (a) Let l be the number obtained by considering the i higher order bits of L, setting it to 0 if $i = 0$.
   (b) Execute $(\mathsf{write}', \mathsf{translate}', \widehat{\mathsf{lab}}') := \mathsf{Eval}(\tilde{C}^{\tau,i}, (\mathsf{DeTranslate}(\mathsf{translate}, \{\hat{r}^{i+1,2l,k}, \hat{r}^{i+1,2l+1,k}\}_{k\in[\kappa]}), \widehat{\mathsf{lab}}))$.
   (c) Parse write' as $(L', \{\alpha^{\mathsf{left},k}, \alpha^{\mathsf{right},k}\}_{k\in[\kappa]})$ and for every $k \in [\kappa]$ if $i < d-1$ update $r^{i+1,2l,k} := \alpha^{\mathsf{left},k}$ and $r^{i+1,2l+1,k} := \alpha^{\mathsf{right},k}$ else update $\tilde{D}^{2l,k} := \alpha^{\mathsf{left},k}$ and $\tilde{D}^{2l+1,k} := \alpha^{\mathsf{right},k}$.
   (d) Update $L = L'$, $\mathsf{translate} := \mathsf{translate}'$ and $\widehat{\mathsf{lab}} := \widehat{\mathsf{lab}}'$.

Figure 8: Formal description of GEval.

5 Security Proof

In this section we prove the UMA-security of the garbled RAM (GData, GProg, GInput, GEval) shown in Sec. 4.

Theorem 5.1 (UMA-security)**.** *Given any OWF and a secure garbling scheme (which can be built from the OWF), our construction is a UMA (unprotected memory access) secure garbled RAM scheme.*

An extension of this theorem provided in Lemma D.1 along with Theorem D.2 allows us to immediately obtain the following corollary.

Corollary 5.2. *Assume one-way functions exist, then a fully secure multi-program GRAM scheme exists (defined in Appendix D).*

Proof Sketch: We now sketch a proof of Theorem 5.1 to give intuition, and then we provide the full proof.

We construct a simulator Sim that produces simulated garbled circuits starting from the last circuit. It proceeds by generating a random-looking output for each $\tilde{C}^{\tau,i}$ by setting translate to be a random key XORed with the corresponding input labels of $\tilde{C}^{\tau,i+1}$ (since we are working backwards, these labels have already been generated), and similarly using random values for write. The main idea is that we perform bookkeeping to keep track of these random values, so that when we simulate the garbled database, we set $\tilde{D}$ to be uniformly random subject to matching the bookkeeping: since the UMA-simulator gets the full access pattern, it knows exactly which locations in memory it should set entries so that they match what was used to mask the translation table.

Then in order to argue that the simulated output is indistinguishable from the real distribution, we define a sequence of hybrids $H^{0,0}, H^{0,1}, \ldots, H^{0,d-1}, H^{1,0}, \ldots, H^{t-1,d-1}$ that replaces circuits from the first circuit onward, where in $H^{\tau,i}$, the circuits $\tilde{C}^{\tau',i'}$ have been simulated for all $(\tau', i') < (\tau, i)$ lexicographically. Moving to $H^{\tau,i}$ from $H^{\tau,i-1}$ (and analogously to $H^{\tau,0}$from $H^{\tau-1,d-1}$), we carefully define subhybrids $H^{\tau,i}_{\mathsf{prf}}$, $H^{\tau,i}_{\mathsf{enc}}$, and $H^{\tau,i}_{\mathsf{circuit}}$, where we first argue that the PRF outputs can be replaced, followed by replacing the labels that won't be decrypted in translate, followed by simulating the circuit now that the dependencies on the unopened labels have been removed.

Proof:

Simulator. Under UMA-security, Sim gets the inputs $1^\kappa, 1^m, 1^t, y, D$ and $\mathsf{MemAccess} = \{(L^\tau, z^{\mathsf{read},\tau}, z^{\mathsf{write},\tau}) : \tau = 1, \ldots, t\}$, where program P executes t CPU steps and outputs y, the initial memory contents are $D \in \{0,1\}^m$ and MemAccess describes the entire memory access throughout all t CPU steps executed.

Overview. The simulator Sim computes the garbled circuits from the last circuit. It uses the knowledge MemAccess to compute the output of each garbled circuit and the nodes that will be visited at each step.

Sim computes every $\tilde{C}^{\tau,i}$ by first computing its output: $\mathsf{translate}, \mathsf{write}, \mathsf{state}$. The table translate consists of random keys xored with the input labels of $\tilde{C}^{\tau,i+1}$ (which are available when computing $\tilde{C}^{\tau,i}$ as $\tilde{C}^{\tau,i+1}$ has been already computed). Such keys are kept in a global file F as they represent the nodes of the memory in a previous step: they will be either used as the output write of some circuit $\tilde{C}^{\tau',i-1}$ for some $\tau' < \tau$, or if no other circuit has visited this node before, they will be part of the garbled data given in input at the beginning.

Notation. Recall that $\overline{\mathsf{lab}}^{\tau,i} = [\overline{\mathsf{lab}}^{\tau,i,\mathsf{read}}, \overline{\mathsf{lab}}^{\tau,i,\mathsf{aux}}]$ and that for a circuit $\tilde{C}^{i,j}$, $\overline{\mathsf{lab}}^{\mathsf{read}} = \{\mathsf{lab}^{\mathsf{left},k,b}, \mathsf{lab}^{\mathsf{right},k,b}\}_{k\in[\kappa],b\in\{0,1\}}$. In the description of the simulator, we abuse the notation and we use $\overline{\mathsf{lab}}^{\tau,i}$ for the keys obtained from the simulator of the garbled circuits. The set set $\overline{\mathsf{lab}}^{\tau,i}$ generated by CircSim will contain only one key per wire.

We use notation $\hat{r}^{i+1,2l,k}\{\tau\}$ the garbled value $\hat{r}^{j+1,2l,k}$ at step τ. The simulator uses auxiliary procedures GenerateWrite shown if Fig. 10 and GenerateTranslate shown in Fig. 9.

Procedure Simulator: $\mathsf{Sim}(1^\kappa, 1^m, 1^t, y, D, \mathsf{MemAccess} = \{L^\tau, z^{\mathsf{read},\tau}, z^{\mathsf{write},\tau}\}_{\tau=1,\ldots,t})$

1. Initialize a file F to store the nodes of the garbled memory visited during the simulated execution.
2. Compute Garbled Circuits.

   For each step $\tau \in \{t-1, \ldots, 0\}$ do.

   - If $\tau = t-1$. $(\tilde{C}^{t-1,d-1}, \overline{\mathsf{lab}}^{t-1,d-1}) \leftarrow \mathsf{CircSim}\left(1^\kappa, C^{\mathsf{final}}, \mathsf{write}, \mathsf{aux}\right)$
     where $\mathsf{aux} = (y, L^{t-1})$; $\mathsf{write} = (\mathsf{GenerateWrite}(t-1, d-1, L^{t-1}), z^{\mathsf{write},t-1})$.
     Else, $(\tilde{C}^{\tau,d-1}, \overline{\mathsf{lab}}^{\tau,d-1}) \leftarrow \mathsf{CircSim}\left(1^\kappa, C^{\mathsf{step}}, \mathsf{write}, \mathsf{translate}, \overline{\mathsf{lab}}^{\tau+1,0,\mathsf{aux}}\right)$ where
     $\mathsf{write} = \mathsf{GenerateWrite}(\tau, d-1, L^\tau)$ and
     $\mathsf{translate} = \mathsf{GenerateTranslate}\ (\tau+1, 0, L^{\tau+1}, \overline{\mathsf{lab}}^{\tau+1,0,\mathsf{read}})$

- For $i \in \{d-2, \ldots, 0\}$. $(\tilde{C}^{\tau,i}, \overline{\mathsf{lab}}^{\tau,i}) \leftarrow \mathsf{CircSim}\left(1^\kappa, C^{\mathsf{nav}}, \mathsf{write}, \mathsf{translate}, \overline{\mathsf{lab}}^{\tau,i+1,\mathsf{aux}}\right)$, where $\mathsf{write} = \mathsf{GenerateWrite}(\tau, i, L^\tau)$ and $\mathsf{translate} = \mathsf{GenerateTranslate}\left(\tau, i+1, L^\tau, \overline{\mathsf{lab}}^{\tau,i+1,\mathsf{read}}\right)$.

Output: $\tilde{C}^{\tau,i}$ for all $\tau \in [t], i \in [d]$.

3. Compute Garbled Inputs. **Output** $\tilde{x} = (\mathsf{translate}, \overline{\mathsf{lab}}^{0,0,\mathsf{aux}})$, where $\mathsf{translate} = \mathsf{GenerateTranslate}\left(0, 1, 0^d, \overline{\mathsf{lab}}^{0,0,\mathsf{read}}\right)$.

4. Compute Garbled Memory. $\forall i \in \{1, \ldots d\}, j \in [2^i], k \in [\kappa]$. Check if in F for the smallest $p > \tau$ such that key $\hat{r}^{i,j,k}\{p\}$ has been set. If so, set $\hat{r}^{i,j,k} = \hat{r}^{i,j,k}\{p\}$ else set $\hat{r}^{i,j,k} \leftarrow \{0,1\}^\kappa$.
Finally, set $\{\hat{D}^{j,k} = \hat{r}^{d,j,k}\}_{j \in [2^d], k \in [\kappa]}$.
Output $\tilde{D} = (\{\hat{r}^{i,j,k}\}_{i \in [d] \setminus \{0\}, j \in [2^i], k \in [\kappa]}, \{\hat{D}^{j,k}\}_{j \in [2^d], k \in [\kappa]})$

Procedure: GenerateTranslate(step τ, level i, index L, input label $\mathsf{lab}^{\mathsf{left},k}, \mathsf{lab}^{\mathsf{right},k}$)

1. Let l be the number obtained by considering the i higher order bits of L, setting it to 0 if $i = 0$.
2. Pick random $r^{\mathsf{left},1}, \ldots, r^{\mathsf{left},k} \leftarrow \{0,1\}^\kappa$ and $r^{\mathsf{right},1}, \ldots, r^{\mathsf{right},k}\{0,1\}^\kappa$ with $k \in [\kappa]$.
3. Set $\hat{r}^{i+1,2l,k}\{\tau\} = r^{\mathsf{left},k}$ and $\hat{r}^{i+1,2l+1,k}\{\tau\} = r^{\mathsf{right},k}$ and store them in the global file F.
4. Output
$$\mathsf{translate} := \left\{ \begin{array}{ll} r^{\mathsf{left},k} \oplus \mathsf{lab}^{\mathsf{left},k}, & r^{\mathsf{left},k} \oplus \mathsf{lab}^{\mathsf{right},k} \\ rand^{\mathsf{left},k}, & rand^{\mathsf{right},k}. \end{array} \right\}_{k \in [\kappa]}$$
where $rand^{\mathsf{right},k}$ and $rand^{\mathsf{left},k}$ are randomly chosen. The rows of translate are randomly permuted.

Figure 9: Procedure GenerateTranslate.

Procedure: GenerateWrite(step τ, level i, index L)

1. Let l be the number obtained by considering the i higher order bits of L.
2. Look up in F for the smallest timestamp $p > \tau$ for which nodes $\hat{r}^{i+1,2l,k}\{p\}$ and $\hat{r}^{i+1,2l+1,k}\{p\}$ with $k \in [\kappa]$ have been already computed. If such p exists set $\alpha^{\mathsf{left},k} := r^{i+1,2l,k}\{p\}$ and $\alpha^{\mathsf{right},k} := r^{i+1,2l+1,k}\{p\}$. (such p exists if in a later step p a translate table was computed to read nodes $(i+1, 2l)$ and $(i+1, 2l+1)$). Else, set $\alpha^{\mathsf{left},k} \leftarrow \{0,1\}^\kappa$ and $\alpha^{\mathsf{right},k} \leftarrow \{0,1\}^\kappa$ with $k \in [\kappa]$.
3. Output $(L, \{\alpha^{\mathsf{left},k}, \alpha^{\mathsf{right},k}\}_{k \in [\kappa]})$.

Figure 10: Procedure GenerateWrite.

Lifetime of a PRF key. Now we analyze the lifetime of any PRF key used in the system. For convenience we recall some key facts about the circuits in our construction and set up some notation that will be handy in understanding the next lemma.

In the garbled memory a node at level i, is computed under some PRF key v and encodes two keys $v^{\mathsf{left}}, v^{\mathsf{right}}$, which are the keys used to compute the left and right child respectively. We say that the PRF key v is *used*, while the PRF keys $v^{\mathsf{left}}, v^{\mathsf{right}}$ are *encoded*.

During the computation, say at step τ, circuit $\tilde{C}^{\tau,i}$ reads the two keys encoded in a node at level i, say $v^{\mathsf{left}}, v^{\mathsf{right}}$; *uses* one of the keys, say v^{left}, to compute $\mathsf{translate}$ and updates the node at level i using a fresh key v'. To update a node means to re-encode the keys v'', v^{right} using the fresh PRF key v' (in our example we are replacing v^{left}). The PRF key v'' is replacing v^{left} and is encoded for the first time in the memory, therefore we say the PRF key v'' was born. Instead, key v^{left}, that is *used* to computed $\mathsf{translate}$ table, at this point is not encoded in the parent anymore and it will be replaced with another key by next circuit $\tilde{C}^{\tau,i+1}$. Therefore, we say that once a key is used to compute $\mathsf{translate}$ table, the key is $\mathtt{dead}$ as it has disappeared from the system.

Key v^{right} instead is re-encoded in the parent node and will be possibly read again in the future.

With this notation in hand we are able to formalize the following Lemma.

Lemma 5.3 (Life of a PRF key)**.** *Part 1. Any PRF key v is used for evaluation in at most two circuits. Part 2. At the time when v dies, v is encoded only in nodes computed with keys that are dead already.*

Part 1. In any step of computation τ, any PRF key v hardwired in circuit $\tilde{C}^{\tau,i}$ is used for PRF evaluation in at most two places:

1. Inside circuit $\tilde{C}^{\tau,i}$ to compute the output write, which is the updated version of nodes at level i.
2. Inside circuit $\tilde{C}^{\tau',i-1}$ for some step $\tau' > \tau$ to compute the translate table to read a node at level i. Circuit $\tilde{C}^{\tau',i-1}$ reads in input v, x and uses v to compute the translate table to read the node computed under key v (specifically, the one that was given in output by $\tilde{C}^{\tau,i}$). Also, circuit $\tilde{C}^{\tau',i-1}$ recomputes the parent node, i.e., the node at level $i-1$, so that it does not encode v.

   This is the last time in which the key v is used. The next circuit $\tilde{C}^{\tau,i}$ will not be aware of v and it will recompute the node at level i with a fresh new key. This prove Part 1 of the lemma.

Part 2. Between time τ and time τ' the key v could have been read, and re-encoded multiple times in the parent node (following the previous example, the parent node lies at level $i-1$).

We know that every time the parent node has been visited (and thus v has been read) it has been readily recomputed under a fresh key. In particular, when $\tilde{C}^{\tau',i-1}$ reads in input v, x and it knows that v is gonna be used next, it re-computes the parent node at level $i-1$ so that it does not encode v anymore. Therefore, at step $(\tau', i-1)$, the only nodes encoding the key v are the ones that were computed under dead keys. This proves Part 2 of the Lemma.

Indistinguishability of the simulation. We prove indistinguishability of the simulation through a sequence of hybrids. Starting from the first circuit, i.e., $\tilde{C}^{0,0}$, in each hybrid we replace a garbled circuit with a simulated circuit. Along the way we replace the nodes visited by the simulated circuits with random nodes.

We say that a pair of nodes (i, j) and $(i, j+1)$ is **visited** at step τ if circuit $\tilde{C}^{\tau,i-1}$ outputs a translate table that uses the memory values $\hat{r}^{i,j,k}$ and $\hat{r}^{i,j+1,k}$ with $k \in [\kappa]$. We say that a node is **random** if all the PRF evaluations are replaced with fresh random values.

The rationale behind the hybrids is the following. Let $H^{\tau,i-1}$ be the experiment where all circuits up to $\tilde{C}^{\tau,i-1}$ have been successfully replaced with simulated circuits and the nodes visited by these circuits have been replaced with random nodes. In hybrid $H^{\tau,i}$ we aim to replace circuit $\tilde{C}^{\tau,i}$ with a simulated one, and claim that the distribution of this experiment is computationally indistinguishable to the distribution of experiment $H^{\tau,i-1}$ by invoking the security of the garbling scheme.

In order to be able to use the security of the garbling scheme we need to argue that the adversary gets only one label for each input wire of circuit $\tilde{C}^{\tau,i}$. Recall that $\tilde{C}^{\tau,i}$ is evaluated using two sets of input labels:

$\overline{\mathsf{lab}}^{\tau,i,\mathsf{aux}}$ and $\overline{\mathsf{lab}}^{\tau,i,\mathsf{read}}$. Set $\overline{\mathsf{lab}}^{\tau,i,\mathsf{aux}}$ is passed directly from circuit $\tilde{C}^{\tau,i-1}$, and contains already one label only for each wire. Due to the security of $\tilde{C}^{\tau,i-1}$ (which we already replaced with a simulated circuit), the adversary cannot learn the other label. Instead, $\overline{\mathsf{lab}}^{\tau,i,\mathsf{read}}$ are passed via translate, and both labels are encrypted in translate using a key derived from the PRF used to compute nodes $(i,j),(i,j+1)$ (for some node j).

The core of the argument is to show that we can replace translate with a table that encrypts only one label per input wire, by first replacing the PRF keys with randomly generated keys and then replacing one of the input labels with random values. Then we can finally replace circuit $\tilde{C}^{\tau,i}$ with a simulated circuit and rely on the security of the garbling scheme. Next, we describe the core hybrid formally.

Hybrid $H^{\tau,i}$. Let $H^{\tau,i-1}$ with $\tau = \{0,\ldots,t-1\}$ and $i = \{1,\ldots,d-1\}$, be the experiment where all circuits till circuit $\tilde{C}^{\tau,i-1}$ have been successfully replaced with simulated circuits. The case where we move between $H^{\tau,0}$ and $H^{\tau-1,d-1}$ is analogous. In hybrid $H^{\tau,i}$ we aim to replace circuit $\tilde{C}^{\tau,i}$ with a simulated one.

Recall that the input labels $\overline{\mathsf{lab}}^{\tau,i,\mathsf{read}}$ used to evaluate circuit $\tilde{C}^{\tau,i}$ are encrypted in the table translate which is given in output by circuit $\tilde{C}^{\tau,i-1}$, and are decrypted using the keys stored in node $\hat{r}^{i,j,k}, \hat{r}^{i,j+1,k}$, for some node j. Note that this node was either never visited, in that case the PRF key used to encrypt this node has never been used in any previous circuit, or it was visited in some step $\tau' < \tau$. In such a case the value $\hat{r}^{i,j,k}, \hat{r}^{i,j+1,k}$ is in the output write of circuit $\tilde{C}^{\tau',i}$ which at this point has been already replaced with simulated circuit. Recall that the outputs of circuit $\tilde{C}^{\tau,i-1}$ are:

1. translate table:

$$\mathsf{translate} := \left\{ \begin{array}{ll} F_{r^{i-1,\lfloor j/2 \rfloor}}(\mathsf{left},k,0) \oplus \mathsf{lab}^{\mathsf{left},k,0}, & F_{r^{i-1,j/2}}(\mathsf{right},k,0) \oplus \mathsf{lab}^{\mathsf{right},k,0}, \\ F_{r^{i-1,\lfloor j/2 \rfloor}}(\mathsf{left},k,1) \oplus \mathsf{lab}^{\mathsf{left},k,1}, & F_{r^{i-1,j/2}}(\mathsf{right},k,1) \oplus \mathsf{lab}^{\mathsf{right},k,1}. \end{array} \right\}_{k\in[\kappa]}$$

   where $\overline{\mathsf{lab}}^{\tau,i,\mathsf{read}} = \left(\mathsf{lab}^{\mathsf{left},k,b}, \mathsf{lab}^{\mathsf{right},k,0}\right)$ with $b = 0,1$. Note that $r^{i-1,\lfloor j/2 \rfloor}$ dies at this step.

2. new node: write = $(L, \hat{r}^{i-1,j,k}, \hat{r}^{i-1,j+1,k}\}_{k\in[\kappa]})$.
   (Note that this node might be used in a later step $\tau'' > \tau$ (if any), to decrypt the translate table computed by a circuit $\tilde{C}^{\tau'',i-1}$, which has not been replaced at this point. Therefore in this hybrid we will not replace write yet: we will replace it in hybrid $H^{\tau'',i}$.)

Sub-hybrid $H^{\tau,i}_{\mathsf{prf}}$. Let $v = r^{i-1,\lfloor j/2 \rfloor}$ be the PRF key used to compute translate given in output $\tilde{C}^{\tau,i-1}$. In this hybrid we want to replace the PRF evaluation computed with the dead key v, with random strings.

Due to Lemma 5.3 we know that any PRF is used for evaluations only within two circuits $\tilde{C}^{\tau',i-1}$ and $\tilde{C}^{\tau,i}$ that, by hypothesis assumption, have been already replaced with simulated circuits. Therefore at this point we can safely replace *their* outputs with values computed with random strings instead of PRF evaluations. However, v was also encoded in previous parents nodes throughout the computation. Fortunately, part 2 of Lemma 5.3 ensures that the PRF key v that we are replacing is only encoded in nodes computed with PRF keys that are already dead at step τ. By hypothesis assumption all nodes computed with dead keys have been already replaced with random nodes (to see why, recall that a key is dead if it has been used to compute translate table in some circuit $\tilde{C}^{\tau'',i}$ with $\tau'' < \tau$. Because all circuits up to $\tilde{C}^{\tau',i-1}$ have been replaced with simulated circuits, it holds that also all the nodes visited by such circuits have been replaced with random nodes).

Therefore at time τ we are guaranteed that the PRF key v is encoded nowhere in the system except in the output of circuits $\tilde{C}^{\tau',i-1}$ and $\tilde{C}^{\tau,i}$.

Thus, in this hybrid we need to replace the key v in the following two places only:

1. the translate table given in output by $\tilde{C}^{\tau,i-1}$, with the new table computed with random values.

$$\mathsf{translate} := \left\{ \begin{array}{ll} r^{\mathsf{left},k,0} \oplus \mathsf{lab}^{\mathsf{left},k,0}, & r^{\mathsf{right},k,0} \oplus \mathsf{lab}^{\mathsf{right},k,0} \\ r^{\mathsf{left},k,1} \oplus \mathsf{lab}^{\mathsf{left},k,1}, & r^{\mathsf{right},k,1} \oplus \mathsf{lab}^{\mathsf{right},k,1}. \end{array} \right\}_{k\in[\kappa]}$$

2. the node write given in output in a previous step $\tau' < \tau$ by circuit $\tilde{C}^{\tau',i}$ with $\mathsf{write} = (L, \{\alpha^{\mathsf{left},k}, \alpha^{\mathsf{right},k}\}_{k\in[\kappa]})$ where $\alpha^{\mathsf{left},k}$ and $\alpha^{\mathsf{right},k}$ are chosen so to allow to decrypt the correct input labels; namely, $\alpha^{\mathsf{left},k} = r^{\mathsf{left},k,r_k^{i,j}}$ and $\alpha^{\mathsf{right},k} = r^{\mathsf{right},k,r_k^{i,j+1}}$. Note this is necessary because in this sub-hybrid circuit $\tilde{C}^{\tau,i}$ is still a real circuit and it is needs to read the correct values for the PRF keys $r^{i,j}, r^{i,j+1}$

Analysis. Assume that there is a distinguisher between the distribution of hybrid $H^{\tau,i-1}$ and $H^{\tau,i}$. Then we can construct an distinguisher D for the PRF. D has access to an oracle $\mathcal{O}$ and has to distinguish whether it is a PRF or a truly random function.

In order to do that, on input $(x, y, P, D, 1^m, 1^t, 1^\kappa)$ D computes that garbled memory and the garbled program, and then replaces the visited nodes and the garbled circuits up to circuit $\tilde{C}^{\tau,i-1}$ with random nodes and simulated garbled circuits. Then, when computing translate given in output by $\tilde{C}^{\tau,i-1}$, it queries the oracle $\mathcal{O}$ on input (left, k, b) and (right, k, b) and obtains keys $r^{\mathsf{left},k,b}$ and $r^{\mathsf{left},k,b}$ for $b = 0, 1$ and $k \in [\kappa]$. Then it replaces the output write of circuit $\tilde{C}^{\tau',i}$ with keys $r^{\mathsf{left},k,r_k^{i,j}}$ and $r^{\mathsf{right},k,r_k^{i,j+1}}$ (i.e., the keys in the nodes should allow to decrypt the correct labels for circuit $\tilde{C}^{\tau,i}$).

Now, if $\mathcal{O}$ is a PRF, then the view generated by D is distributed identically to hybrid $H^{\tau,i-1}$, if instead $\mathcal{O}$ is it a random oracle then the view generated by D is distributed as hybrid $H^{\tau,i}$. Consequently, due to the security property of the PRF, we conclude that the two hybrids are indistinguishable.

Sub-hybrid $H^{\tau,i}_{\mathsf{enc}}$. In this sub-hybrid we replace the labels that are not decrypted with random labels. Namely, we compute the translate table given in output by $\tilde{C}^{\tau,i-1}$ with the following:

$$\mathsf{translate} := \left\{ \begin{array}{ll} r^{\mathsf{left},k} \oplus \mathsf{lab}^{\mathsf{left},k}, & r^{\mathsf{left},k} \oplus \mathsf{lab}^{\mathsf{right},k} \\ rand^{\mathsf{left},k}, & rand^{\mathsf{right},k}. \end{array} \right\}_{k\in[\kappa]}$$

where the two raws are given in random order. Because the keys used to encrypt the labels are random strings, and because for each raw the adversary has only access to one of the keys, the other label is information theoretically hidden. The translate table given in output this hybrid is identically distributed to the one given in output in the hybrid $H^{\tau,i}_{\mathsf{prf}}$.

Sub-hybrid $H^{\tau,i}_{\mathsf{circuit}}$. In this sub-hybrid we replace circuit $\tilde{C}^{\tau,i}$ with a simulated one. Due to the security of the underlying garbled scheme, $H^{\tau,i}_{\mathsf{circuit}}$ is computationally indistinguishable from sub-hybrid $H^{\tau,i}_{\mathsf{enc}}$.

Now note that $H^{\tau,i}_{\mathsf{enc}} = H^{\tau,i}$. Therefore $H^{\tau,i}$ is indistinguishable from $H^{\tau,i-1}$. This completes the proof.

□

Acknowledgments

Work supported in part by NSF grants 09165174, 1065276, 1118126 and 1136174, US-Israel BSF grant 2008411, OKAWA Foundation Research Award, IBM Faculty Research Award, Xerox Faculty Research Award, B. John Garrick Foundation Award, Teradata Research Award, and Lockheed-Martin Corporation

Research Award. This material is based upon work supported by the Defense Advanced Research Projects Agency through the U.S. Office of Naval Research under Contract N00014 -11 -1-0392. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[AIK10] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *ICALP (1)*, volume 6198 of *Lecture Notes in Computer Science*, pages 152–163. Springer, 2010.

[AIK11] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. In Rafail Ostrovsky, editor, *52nd Annual Symposium on Foundations of Computer Science*, pages 120–129, Palm Springs, California, USA, October 22–25, 2011. IEEE Computer Society Press.

[AIKW13] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate or how to compress garbled circuits keys. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 166–184, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Berlin, Germany.

[Ajt10] Miklós Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In Leonard J. Schulman, editor, *42nd Annual ACM Symposium on Theory of Computing*, pages 181–190, Cambridge, Massachusetts, USA, June 5–8, 2010. ACM Press.

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Berlin, Germany.

[BGT14] Nir Bitansky, Sanjam Garg, and Sidharth Telang. Succinct randomized encodings and their applications. Cryptology ePrint Archive, Report 2014/771, 2014. http://eprint.iacr.org/.

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

[CHJV14] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and ram programs. Cryptology ePrint Archive, Report 2014/769, 2014. http://eprint.iacr.org/.

[CP13] Kai-Min Chung and Rafael Pass. A simple ORAM. Cryptology ePrint Archive, Report 2013/243, 2013. http://eprint.iacr.org/2013/243.

[CR73] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In Yuval Ishai, editor, *TCC 2011: 8th Theory of Cryptography*

Conference, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163, Providence, RI, USA, March 28–30, 2011. Springer, Berlin, Germany.

[FN93] Amos Fiat and Moni Naor. Broadcast encryption. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 480–491, Santa Barbara, CA, USA, August 22–26, 1993. Springer, Berlin, Germany.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, Maryland, USA, May 31 – June 2, 2009. ACM Press.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 1–17, Athens, Greece, May 26–30, 2013. Springer, Berlin, Germany.

[GGH+13b] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual Symposium on Foundations of Computer Science*, pages 40–49, Berkeley, CA, USA, October 26–29, 2013. IEEE Computer Society Press.

[GGH+13c] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In Emiliano De Cristofaro and Matthew Wright, editors, *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th Annual Symposium on Foundations of Computer Science*, pages 464–479, Singer Island, Florida, October 24–26, 1984. IEEE Computer Society Press.

[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, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422, Copenhagen, Denmark, May 11–15, 2014. Springer, Berlin, Germany.

[GHRW14a] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited, part I. Cryptology ePrint Archive, Report 2014/082, 2014. http://eprint.iacr.org/2014/082.

[GHRW14b] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *Annual Symposium on Foundations of Computer Science, FOCS 2014*, 2014.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 536–553, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Berlin, Germany.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011: 38th International Colloquium on Automata, Languages and Programming, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 576–587, Zurich, Switzerland, July 4–8, 2011. Springer, Berlin, Germany.

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

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 182–194, New York City,, New York, USA, May 25–27, 1987. ACM Press.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Yuval Rabani, editor, *23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 143–156, Kyoto, Japan, January 17–19, 2012. ACM-SIAM.

[LO13a] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 377–396, Tokyo, Japan, March 3–6, 2013. Springer, Berlin, Germany.

[LO13b] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734, Athens, Greece, May 26–30, 2013. Springer, Berlin, Germany.

[LO14] Steve Lu and Rafail Ostrovsky. Garbled RAM revisited, part II. Cryptology ePrint Archive, Report 2014/083, 2014. http://eprint.iacr.org/2014/083.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[LP14] Huijia Lin and Rafael Pass. Succinct garbling schemes and applications. Cryptology ePrint Archive, Report 2014/766, 2014. http://eprint.iacr.org/.

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, Santa Barbara, CA, USA, August 16–20, 1987. Springer, Berlin, Germany.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th Annual ACM Symposium on Theory of Computing*, pages 294–303, El Paso, Texas, USA, May 4–6, 1997. ACM Press.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd Annual ACM Symposium on Theory of Computing*, pages 514–523, Baltimore, Maryland, USA, May 14–16, 1990. ACM Press.

[Ost92] Rafail Ostrovsky. *Software Protection and Simulation On Oblivious RAMs*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1992.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[SvDS+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 13: 20th Conference on Computer and Communications Security*, pages 299–310, Berlin, Germany, November 4–8, 2013. ACM Press.

[WHC+14] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, abhi shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. Cryptology ePrint Archive, Report 2014/671, 2014. http://eprint.iacr.org/2014/671.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.

A Other Related Work

In this section we highlight some of the closely related works.

Interactive Solutions. The first interactive RAM solution proposed for this problem was known as Oblivious RAM and was introduced by Goldreich and Ostrovsky [Gol87, Ost90, Ost92, GO96]. Oblivious RAM (ORAM) allowed the client to interactively perform secure CPU steps with the cloud so that the cloud learns nothing about the data or access pattern. Further works with best-known overhead include those of Kushilevitz et al. [KLO12] (with constant client memory), Goodrich and Mitzenmacher [GM11] (with poly client memory), and Stefanov et al. [SvDS+13] (with larger block sizes).

The work of Ostrovsky-Shoup [OS97] introduced a secure compiler for RAM programs (via ORAM) for general secure RAM computation. The idea behind this was to for the two or more parties to securely emulate the ORAM client which was modeled as a CPU circuit using secure circuit computation techniques. Gordon et al. [GKK+12] demonstrated an efficient realization and implemented this compiler, using the example of secure binary search. The best theoretical overhead for secure RAM computation comes from a compilation using a two-server ORAM devised by Lu and Ostrovsky [LO13a], and the fastest implementation is due to [WHC+14].

A tantalizing approach would be to combine the techniques of FHE with ORAM in order to remove interaction. Indeed, combining FHE with interactive RAM computation to reduce communication or rounds has been explored in works such as [GGH+13c]. However, as mentioned above, the evaluator would run (under FHE) the ORAM-compiled CPU circuit and get an encrypted location as output. Unless this location is decrypted via interaction, this location could only be used as an encrypted selector against the entire encrypted database (or some portion of it if some pre-specified data structures are used). But since this is done at every CPU step, the evaluation time would be prohibitively large for general applications.

Garbling Beyond Boolean Circuits. We consider models that were explored beyond Boolean circuits when garbling. Many real-world computations fall into the category of arithmetic computations, so a natural solution to these sitautions would be to use arithmetic circuits instead. The first arithmetic circuit garbling scheme was introduced by Applebaum et al. [AIK11], motivated by the problem of garbling circuits of this variant.

Goldwasser et al. [GKP+13] created a reusable, non-interactive method for encrypted computation on TM programs. This can be viewed as an extending reusabled garbled circuits which critically bypasses the downsides of the circuit model. We want to explore non-interactive garbling-style solutions for RAM

programs (which has additional merits on top of TMs, such as in the case of binary search), with ideally an overhead that is only *polylogarithmic* in n.

Garbled RAM was first introduced by Lu and Ostrovsky [LO13b] in which they constructed a GRAM scheme with poly-log overhead from any one-way function, but required a circularity assumption for their proof. This assumption was subsequently removed at a cost by the work of Gentry et al. [GHL+14, GHRW14a, LO14] which presented two solutions: one which achieves poly-log overhead but assumes the existence of identity-based encryption, and the other which assumes only one-way functions but only achieves $|D|^\epsilon$ overhead where $|D|$ is the size of the database. Recently, the work of Gentry et al. [GHRW14b] constructed two new GRAM schemes that could be applied to outsourcing computation. The first scheme assumes indistinguishability obfuscation [GGH+13b] based on multilinear maps [GGH13a], one-way functions, and simulation-sound non-interactive zero knowledge, and is reusable but does not support a persistent database across multiple programs (as the previous schemes did) so a fresh database would need to be garbled each time as part of input. Their second scheme is reusable, supports a persistent database across multiple programs, and is compact (where the garbled programs are independent of the program running time), but is based on stronger reasonably-conjectured assumptions.

RAM Obfuscation. Recently, works have considered the notion of obfuscating RAM programs [GHRW14b, CHJV14, BGT14, LP14]. Just as circuit obfuscation requires stronger assumptions than garbled circuits, these new results achieve RAM obfuscation, but also require stronger assumptions than just one-way functions. We highlight that our GRAM construction has the distinction that it can run multiple programs, garbled independently from the database, on a persistent database where only in garbling the inputs is the order determined. RAM obfuscation on the other hand typically do not satisfy these properties, though they have the benefit of being reusable and can be ran on arbitrary inputs.

B Oblivious RAM

We review Oblivious RAM (ORAM), which was first introduced by Goldreich and Ostrovsky [Gol87, Ost90, Ost92, GO96]. This can be thought of as a compiler that encodes the memory and program into a special format that does not reveal the access pattern or data contents during an execution. We use the notation of ORAM from [GHL+14] and refer the reader to [Ost92] for a detailed account of ORAM. Since we are aiming for multi-program security, we slightly extend the definition of Oblivious RAM to ensure we can go from program to program. As before, we assume there is some security parameter κ and all other sizes are at most some polynomial in this parameter.

Syntax. A *Oblivious RAM* scheme consists of two procedures $(\mathsf{OData}, \mathsf{OProg})$ with syntax:

- $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D)$: Given a security parameter κ and memory $D \in \{0,1\}^m$ as input, OData outputs the encoded memory D^* and encoding key s^*.
- $P^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log m}, 1^t, P)$: Given a security parameter κ, a memory size m, and a program P that runs in time t, OProg outputs an oblivious program P^* that can access D^* as RAM and takes two inputs x and s^*.

Efficiency. We require that the run-time of OData should be $m \cdot \mathsf{polylog}(m) \cdot \mathsf{poly}(\kappa)$, and the run-time of OProg should be $t \cdot \mathsf{poly}(\kappa) \cdot \mathsf{polylog}(m)$. Finally, the oblivious program P^* itself should run in time $t' = t \cdot \mathsf{poly}(\kappa) \cdot \mathsf{polylog}(m)$. Both the new memory size $m' = |D^*|$ and the running time t' should be efficiently computable from m, t, and κ.

Correctness. Let $P_1, \ldots, P_\ell$ be programs running in polynomial times $t_1, \ldots, t_\ell$ on memory D of size m. Let $x_1, \ldots, x_\ell$ be the inputs and κ be a security parameter. Then we require that:

$$\Pr[(P_1^*(x_1, s^*), \ldots, P_\ell^*(x_\ell, s^*))^{D^*} = (P_1(x_1), \ldots, P_\ell(x_\ell))^D] = 1$$

where $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D)$, $P_i^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log m}, 1^t, P_i)$ and $(P_1^*(x_1, s^*), \ldots, P_\ell^*(x_\ell, s^*))^{D^*}$ indicates running the ORAM programs on D^* sequentially.

Security. For security, we require that there exists a PPT simulator Sim such that for any sequence of programs $P_1, \ldots, P_\ell$, initial memory data $D \in \{0,1\}^m$, and inputs $x_1, \ldots, x_\ell$ we have that:

$$(D^*, \mathsf{MemAccess}) \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\kappa, 1^m, \{1^{t_i}, y_i\}_{i=1}^\ell)$$

where $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$, $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D)$, and $\mathsf{MemAccess}$ corresponds to the access pattern of the CPU-step circuits during the sequential execution of the oblivious programs $(P_1^*(x_1, s^*), \ldots, P_\ell^*(x_\ell, s^*))^{D^*}$.

C Full Single-Program Security

Constructing full GRAM from UMA-secure GRAM was previously proven in [GHL+14]. Given our slightly stronger definition where GProg is decoupled from GData, we re-prove the result under these stronger conditions for the sake of completeness.

Before moving to multi-program security, we first define what a *fully secure single-program GRAM* should satisfy. The only difference is in the security definition (as defined in Section 3), namely that the simulator no longer has access to D or $\mathsf{MemAccess}$.

Security. For security, we require that there exists a PPT simulator Sim such that for any program P running in time t, initial memory data $D \in \{0,1\}^m$ and input x we have that:

$$(\tilde{D}, \tilde{P}, \tilde{x}) \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\kappa, 1^m, 1^t, y).$$

We prove the following theorem.

Theorem C.1. *Assume there exists a UMA-secure single-program GRAM scheme and an ORAM scheme (both of which could be efficiently constructed from OWFs). Then there exists a fully secure single-program GRAM scheme. Moreover, we give a black-box construction of one given a UMA-secure GRAM and ORAM scheme.*

Proof. We first give the construction of the scheme itself and then provide a construction of an appropriate simulator to prove security. Let $(\mathsf{GData}, \mathsf{GProg}, \mathsf{GInput}, \mathsf{GEval})$ be a UMA-secure single-program GRAM and let $(\mathsf{OData}, \mathsf{OProg})$ be an ORAM scheme. We construct a new single-program GRAM scheme $(\widehat{\mathsf{GData}}, \widehat{\mathsf{GProg}}, \widehat{\mathsf{GInput}}, \widehat{\mathsf{GEval}})$ as follows:

- $\widehat{\mathsf{GData}}(1^\kappa, D)$: Execute $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D)$ followed by $(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D^*)$. Output $\widehat{D} = \tilde{D}$ and $\widehat{s} = (s, s^*)$.
- $\widehat{\mathsf{GProg}}(1^\kappa, 1^{\log m}, 1^t, P)$: Execute $P^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log m}, 1^t, P)$ followed by $(\tilde{P}, s^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log m'}, 1^{t'}, P^*)$. Output $\widehat{P} = \tilde{P}$, $\widehat{s^{in}} = s^{in}$ (note that this contains the garbled wire labels for s^* since P^* takes two inputs). Also note that we use m' and t' since ORAM will increase the running time and memory size by some poly-logarithmic overhead.
- $\widehat{\mathsf{GInput}}(1^\kappa, x, \widehat{s^{in}}, \widehat{s})$: Parse $\widehat{s} = (s, s^*)$ and set $x^+ = (x, s^*)$ which is valid input for P^*. Execute $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x^+, \widehat{s^{in}}, s)$, and output $\widehat{x} = \tilde{x}$.
- $\widehat{\mathsf{GEval}}^{\widehat{D}}(\widehat{P}, \widehat{x})$: Execute $y \leftarrow \mathsf{GEval}^{\widehat{D}}(\widehat{P}, \widehat{x})$ and output y.

We show that the construction above given by $(\widehat{\mathsf{GData}}, \widehat{\mathsf{GProg}}, \widehat{\mathsf{GInput}}, \widehat{\mathsf{GEval}})$ is a fully secure single-program GRAM scheme.

Correctness. Our goal is to demonstrate that

$$\Pr[\widehat{\mathsf{GEval}}^{\widehat{D}}(\widehat{P},\widehat{x}) = P^D(x)] = 1$$

where $(\widehat{D},\widehat{s}) \leftarrow \widehat{\mathsf{GData}}(1^\kappa, D)$, $(\widehat{P},\widehat{s^{in}}) \leftarrow \widehat{\mathsf{GProg}}(1^\kappa, 1^{\log m}, 1^t, P)$, $\widehat{x} \leftarrow \widehat{\mathsf{GInput}}(1^\kappa, x, \widehat{s^{in}}, \widehat{s})$.

By definition, $\widehat{\mathsf{GEval}}^{\widehat{D}}(\widehat{P},\widehat{x}) = \mathsf{GEval}^{\tilde{D}}(\tilde{P},\tilde{x})$. By the correctness of the UMA-secure GRAM scheme, we have that $\mathsf{GEval}^{\tilde{D}}(\tilde{P},\tilde{x}) = P^{*D^*}(x, s^*)$. Finally, by the correctness of the ORAM scheme, $P^{*D^*}(x, s^*) = P^D(x)$.

Security. For any program P, database D, and input x, let $\text{REAL}^{D,P,x}$ define the following distribution:

$$\begin{aligned}\text{REAL}^{D,P,x} = \{(\widehat{D},\widehat{P},\widehat{x}) :& (\widehat{D},\widehat{s}) \leftarrow \widehat{\mathsf{GData}}(1^\kappa, D),\\ & (\widehat{P},\widehat{s^{in}}) \leftarrow \widehat{\mathsf{GProg}}(1^\kappa, 1^{\log m}, 1^t, P),\\ & \widehat{x} \leftarrow \mathsf{GInput}(1^\kappa, x, \widehat{s^{in}}, \widehat{s})\}\end{aligned}$$

Our goal is to construct a simulator Sim such that for all D, P, x and $y = P^D(x)$, $\text{REAL}^{D,P,x} \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\kappa, 1^m, 1^t, y)$. We let OSim be the ORAM simulator, and USim be the simulator for the UMA-secure GRAM scheme. The procedure $\mathsf{Sim}(1^\kappa, 1^m, 1^t, y)$ proceeds as follows.

1. Compute $(D^*, \mathsf{MemAccess}) \leftarrow \mathsf{OSim}(1^\kappa, 1^m, 1^t, y)$.
2. Compute $(\tilde{D},\tilde{P},\tilde{x}) \leftarrow \mathsf{USim}(1^\kappa, 1^{m'}, 1^{t'}, y, D^*, \mathsf{MemAccess})$.
3. Output $(\widehat{D},\widehat{P},\widehat{x}) = (\tilde{D},\tilde{P},\tilde{x})$.

We now prove the output of the simulator is computationally indistinguishable from the real distribution. For any D, P, x, we define a series of hybrid distributions $\mathbf{Hyb}_0, \mathbf{Hyb}_1, \mathbf{Hyb}_2$ with $\mathbf{Hyb}_0 = \text{REAL}^{D,P,x}$, and $\mathbf{Hyb}_2 = \mathsf{Sim}(1^\kappa, 1^m, 1^t, y)$, and argue that for $i = 0, 1$ we have $\mathbf{Hyb}_i \overset{\text{comp}}{\approx} \mathbf{Hyb}_{i+1}$.

- $\mathbf{Hyb}_0$: This is the real distribution $\text{REAL}^{D,P,x}$.
- $\mathbf{Hyb}_1$: Use the correctly generated (D^*, s^*) from $\widehat{\mathsf{GData}}$ and P^* from $\widehat{\mathsf{GProg}}$ and execute $P^{*D^*}(x, s^*)$ to obtain y and a sequence of memory accesses $\mathsf{MemAccess}$. Run $(\tilde{D},\tilde{P},\tilde{x}) \leftarrow \mathsf{USim}(1^\kappa, 1^{m'}, 1^{t'}, y, D^*, \mathsf{MemAccess})$ and output $(\widehat{D},\widehat{P},\widehat{x}) = (\tilde{D},\tilde{P},\tilde{x})$. Formally,

$$\begin{aligned}\mathbf{Hyb}_1 = \{(\widehat{D},\widehat{P},\widehat{x}) :& (D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D), P^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log m}, 1^t, P),\\ & y \leftarrow P^{*D^*}(x, s^*) \text{ and induces } \mathsf{MemAccess},\\ & (\tilde{D},\tilde{P},\tilde{x}) \leftarrow \mathsf{USim}(1^\kappa, 1^{m'}, 1^{t'}, y, D^*, \mathsf{MemAccess}),\\ & (\widehat{D},\widehat{P},\widehat{x}) = (\tilde{D},\tilde{P},\tilde{x})\}.\end{aligned}$$

- $\mathbf{Hyb}_2$: This is the simulated distribution $\mathsf{Sim}(1^\kappa, 1^m, 1^t, y)$.

We now show that adjacent hybrid distributions are computationally indistinguishable.

$\mathbf{Hyb}_0 \overset{\text{comp}}{\approx} \mathbf{Hyb}_1$: Let $\mathcal{A}$ be a PPT distinguisher between these two distributions for some D, P, x. We construct an algorithm $\mathcal{B}$ that breaks the UMA-security of the underlying GRAM scheme that proceeds as follows. First, $\mathcal{B}$ runs $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D), P^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log m}, 1^t, P)$ and declares

$P^*, D^*, x^+ = (x, s^*)$ as the challenge program, database, and input for the UMA-security GRAM game. The UMA-security challenger then outputs $(\tilde{D}', \tilde{P}', \tilde{x}')$ and $\mathcal{B}$ must output a guess whether it is real or simulated. In order to do so, $\mathcal{B}$ sets $(\widehat{D}', \widehat{P}', \widehat{x}') = (\tilde{D}', \tilde{P}', \tilde{x}')$ and forwards this as the challenge to $\mathcal{A}$. $\mathcal{B}$ then outputs the same guess as $\mathcal{A}$.

Observe that if the UMA challenger outputs the real values, then $(\widehat{D}', \widehat{P}', \widehat{x}')$ is distributed identically as if it were generated from $\mathbf{Hyb}_0$, and if the UMA challenger outputs simulated values, $(\widehat{D}', \widehat{P}', \widehat{x}')$ is distributed identically as if it were generated from $\mathbf{Hyb}_1$. Therefore, $\mathcal{A}$ distinguishes with the same probability as $\mathcal{B}$, which is negligible by the UMA-security of the underlying GRAM scheme.

$\mathbf{Hyb}_1 \overset{\text{comp}}{\approx} \mathbf{Hyb}_2$: Let $\mathcal{A}$ be a PPT distinguisher between these two distributions for some D, P, x. We construct an algorithm $\mathcal{B}$ that breaks the security of the underlying ORAM scheme that proceeds as follows. First, $\mathcal{B}$ announces D, P, x as the challenge program, database, and input for the ORAM security game. The ORAM challenger then outputs $(D^{*\prime}, \mathsf{MemAccess}')$ which is either real or simulated. Then, $\mathcal{B}$ computes $y = P^D(x)$ and runs $(\tilde{D}', \tilde{P}', \tilde{x}') \leftarrow \mathsf{USim}(1^\kappa, 1^{m'}, 1^{t'}, y, D^{*\prime}, \mathsf{MemAccess}')$. Next, $\mathcal{B}$ sets $(\widehat{D}', \widehat{P}', \widehat{x}') = (\tilde{D}', \tilde{P}', \tilde{x}')$ and forwards this to $\mathcal{A}$. $\mathcal{B}$ then outputs the same guess as $\mathcal{A}$.

Observe that if the ORAM challenger outputs the real values, then $(\widehat{D}', \widehat{P}', \widehat{x}')$ is distributed identically as if it were generated from $\mathbf{Hyb}_1$, and if the ORAM challenger outputs simulated values, then $(\widehat{D}', \widehat{P}', \widehat{x}')$ is distributed identically as if it were generated from $\mathbf{Hyb}_2$. Therefore, $\mathcal{A}$ distinguishes with the same probability as $\mathcal{B}$, which is negligible by the security of the underlying ORAM scheme.

□

D Full Multi-program Security

For multiple programs, we give a more formal treatment of constructing full GRAM from UMA-secure GRAM than was provided in [GHL+14]. In particular, our definition allows for the garbling of the programs independently of each other, and so for completeness we formally state and prove that this is still compatible with the existing ORAM compliation technique.

We define what a *secure multi-program GRAM* is, and as a warmup construct a UMA-secure one and show how to extend it to full security. The syntax remains largely the same as in the single-program GRAM case, except now we have two additional keys: w and s^{out} which we highlight with red in the syntax definition. The w key is output by the GData algorithm and is generated once and the same one is used for each GInput, whereas s^{out} is an evolving key that is initially equal to s and is updated by GProg. Note that GData and GProg are still decoupled and programs are garbled independently of each other, and only tied together by the use of GInput. The GInput procecure takes the most recent s^{out} as its analogous s in the single-program case, which enforces the ordering, and it also takes the fixed w from GData. Later, w shall play the role of the ORAM key which remains constant throughout multiple programs, though in our UMA construction we leave it null.

Syntax. A *(UMA) secure multi-program GRAM* scheme consists of four procedures: (GData, GProg, GInput, GEval) with the following syntax:

- $(\tilde{D}, s, w) \leftarrow \mathsf{GData}(1^\kappa, D)$: Given a security parameter 1^κ and memory $D \in \{0,1\}^m$ as input GData outputs the garbled memory $\tilde{D}$. The auxiliary key w is used for garbling multiple programs.
- $(\tilde{P}, s^{in}, s^{out}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log m}, 1^t, P)$: Takes the description of a RAM program P with memory-size m as input. It then outputs a garbled program $\tilde{P}$ and an input-garbling-key s^{in}. The role of s^{out} is to enable garbling multiple programs.
- $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x, s^{in}, s, w)$: Takes as input $x \in \{0,1\}^n$ and and an input-garbling-key s^{in}, a garbled database key s, an auxiliary key w, and outputs a garbled-input $\tilde{x}$.

- $y = \mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x})$: Takes a garbled program $\tilde{P}$, garbled input $\tilde{x}$ and garbled memory data $\tilde{D}$ and output a value y. We model GEval itself as a RAM program that can read and write to arbitrary locations of its memory initially containing $\tilde{D}$.

Efficiency. We require the run-time of GProg and GEval to be $t \cdot \mathsf{poly}(\kappa) \cdot \mathsf{polylog}(m)$, which also serves as the bound on the size of the garbled program $\tilde{P}$. Moreover, we require that the run-time of GData should be $m \cdot \mathsf{polylog}(m) \cdot \mathsf{poly}(\kappa)$, which also serves as an upper bound on the size of $\tilde{D}$. Finally the running time of GInput is required to be $n \cdot \mathsf{poly}(\kappa)$.

To define the correctness and security requirements of garbled RAMs, we set forth the following notation. Let $P_1, \ldots, P_\ell$ be any sequence of programs with polynomially-bounded run-times $t_1, \ldots, t_\ell$. Let $D \in \{0,1\}^m$ be any initial memory data, let $x_1, \ldots, x_\ell$ be inputs and $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$ be the outputs given by the sequential execution of the programs on D. This plain execution also induces access pattern MemAccess.

In order to tie together the sequence of programs and their inputs, we let $(\tilde{D}_0, s_0^{out} = s, w) \leftarrow \mathsf{GData}(1^\kappa, D)$, and for $i = 1 \ldots \ell$, $(\tilde{P}_i, s_i^{in}, s_i^{out}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log m}, 1^{t_i}, P_i)$, $\tilde{x}_i \leftarrow \mathsf{GInput}(1^\kappa, x_i, s_i^{in}, s_{i-1}^{out}, w)$. Finally, for $i = 1 \ldots \ell$, we consider the outputs obtained by sequential execution: $y_i' = \mathsf{GEval}^{\tilde{D}_{i-1}}(\tilde{P}_i, \tilde{x}_i)$, where $\tilde{D}_i$ is the database after the i^{th} execution.

Correctness. For correctness, we require that:

$$\Pr[(y_1', \ldots, y_\ell') = (y_1, \ldots, y_\ell)] = 1.$$

(UMA) Security. For full security, we require that there exists a PPT simulator Sim such that for any sequence of programs $P_1, \ldots, P_\ell$, initial memory data $D \in \{0,1\}^m$ and inputs $x_1, \ldots, x_\ell$ we have that:

$$(\tilde{D}, \{\tilde{P}_i, \tilde{x}_i\}_{i=1}^\ell) \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\kappa, 1^m, \{1^{t_i}, y_i\}_{i=1}^\ell).$$

For UMA-security, we additionally provide the simulator with $(D, \mathsf{MemAccess})$.

D.1 UMA-secure Multi-Program GRAM Scheme

We describe how to augment our UMA single-program GRAM $(\mathsf{GData}, \mathsf{GProg}, \mathsf{GInput}, \mathsf{GEval})$ described in Section 4 to a UMA multi-program GRAM $(\mathsf{GData}', \mathsf{GProg}', \mathsf{GInput}', \mathsf{GEval}')$. This is described in Figure 11.

- Data Garbling: $(\tilde{D}, s, w) \leftarrow \mathsf{GData}'(1^\kappa, D)$. This proceeds identically to GData and sets $w = \perp$.
- Program Garbling: $(\tilde{P}, s^{in}, s^{out}) \leftarrow \mathsf{GProg}'(1^\kappa, 1^{\log m}, 1^t, P)$. This proceeds identically to GProg except at the end, we set the s^{out} to be the final root key u_t.
- Input Garbling: $\tilde{x} \leftarrow \mathsf{GInput}'(1^\kappa, x, s^{in}, s, w)$. This proceeds identically to GInput since $w = \perp$.
- Garbled Evaluation: $y \leftarrow \mathsf{GEval}'^{\tilde{D}}(\tilde{P}, \tilde{x})$. This proceeds identically to GEval. Observe that at the end of the program, the joint distribution of the new garbled database $\tilde{D}'$ along with s^{out} is identical to that of $\tilde{D}$ and s.

Figure 11: A UMA-secure multi-program GRAM.

We state the following lemma.

Lemma D.1. *The construction of* $(\mathsf{GData}', \mathsf{GProg}', \mathsf{GInput}', \mathsf{GEval}')$ *in Figure 11 is a UMA-secure multi-program GRAM scheme. In particular, assume one-way functions exist, and let the security parameter be*

κ. Then, for any initial RAM contents D of size m, a program P that runs on D in at most t CPU steps, there exists a UMA-secure multi-program Garbled RAM scheme with $\mathsf{poly}(\kappa, \log m, \log t)$ *overhead in the storage size of the garbled memory contents and in the size of the program P and its running time.*

Proof Sketch. The correctness of this scheme follows by combining the fact that the first program executes correctly, and then after the execution of the first garbled program, the joint distribution of the resulting $\tilde{D}'$ and subsequent programs is identical to one in which the first program never existed and $\tilde{D}'$ was generated directly from GData' called on input D', the plain database that would have resulted from the execution of the first plain program.

The simulator for this construction is nearly identical to our UMA single-program GRAM simulator. This is because of the way we connect adjacent programs together: the root key used in the final step of one program is output by GProg as s^{out}, which is then passed as labels into the next garbled program's first circuit. The passing of labels between garbled circuits across the boundary of two programs is therefore identical to between two adjacent steps within a single program, with the only difference being that the read location L is set to 0 and the state is set to a new input. The proof proceeds similarly to our UMA single-program case and we skip the details here.

□

D.2 From UMA to Fully Secure Multi-Program GRAM

We now show how to combine ORAM with UMA-secure multi-program GRAM to obtain fully secure multi-program GRAM. This construction and proof is analogous to our single-program case, though we need to carefully weave together how the ORAM key s^* interacts with the GRAM keys w and s/s^{out}.

We prove the following theorem.

Theorem D.2. *Assume there exists a UMA-secure multi-program GRAM scheme and an ORAM scheme (both of which could be efficiently constructed from OWFs). Then there exists a fully secure multi-program GRAM scheme. Moreover, we give a black-box construction of one given a UMA-secure multi-program GRAM and ORAM scheme.*

Proof. We construct the new GRAM scheme in a black-box manner as follows. Let $(\mathsf{GData}, \mathsf{GProg}, \mathsf{GInput}, \mathsf{GEval})$ be a UMA-secure multi-program GRAM and let $(\mathsf{OData}, \mathsf{OProg})$ be an ORAM scheme. We construct a new multi-program GRAM scheme $(\widehat{\mathsf{GData}}, \widehat{\mathsf{GProg}}, \widehat{\mathsf{GInput}}, \widehat{\mathsf{GEval}})$ as follows:

- $\widehat{\mathsf{GData}}(1^\kappa, D)$: Execute $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D)$ followed by $(\tilde{D}, s, w) \leftarrow \mathsf{GData}(1^\kappa, D^*)$. Output $\widehat{D} = \tilde{D}$ and $\widehat{s} = s$ and $\widehat{w} = (w, s^*)$. Note that unlike in the single-program case, we tie together s^* with the unchanging auxiliary key as opposed to the evolving database key.
- $\widehat{\mathsf{GProg}}(1^\kappa, 1^{\log m}, 1^t, P)$: Execute $P^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log m}, 1^t, P)$ followed by $(\tilde{P}, s^{in}, s^{out}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log m'}, 1^{t'}, P^*)$. Output $\widehat{P} = \tilde{P}$, $\widehat{s^{in}} = s^{in}$, $\widehat{s^{out}} = s^{out}$ (note that this contains the garbled wire labels for s^* since P^* takes two inputs).
- $\widehat{\mathsf{GInput}}(1^\kappa, x, \widehat{s^{in}}, \widehat{s}, \widehat{w})$: Parse $\widehat{s} = s$, $\widehat{w} = (w, s^*)$ and set $x^+ = (x, s^*)$ which is valid input for P^*. Execute $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x^+, \widehat{s^{in}}, s, w)$, and output $\widehat{x} = \tilde{x}$.
- $\widehat{\mathsf{GEval}}^{\widehat{D}}(\widehat{P}, \widehat{x})$: Execute $y \leftarrow \mathsf{GEval}^{\widehat{D}}(\widehat{P}, \widehat{x})$ and output y.

We show that the construction above given by $(\widehat{\mathsf{GData}}, \widehat{\mathsf{GProg}}, \widehat{\mathsf{GInput}}, \widehat{\mathsf{GEval}})$ is a fully secure multi-program GRAM scheme. For the remainder of the proof, we consider the following notation. Let $P_1, \ldots, P_\ell$ be any sequence of programs with polynomially-bounded run-times $t_1, \ldots, t_\ell$. Let $D \in \{0,1\}^m$ be any initial memory data, let $x_1, \ldots, x_\ell$ be inputs and $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$ be the outputs

given by the sequential execution of the programs on D. Let $(\widehat{D}_0, \widehat{s_0^{out}} = \widehat{s}, \widehat{w}) \leftarrow \widehat{\mathsf{GData}}(1^\kappa, D)$, and for $i = 1 \dots \ell$: $(\widehat{P}_i, \widehat{s_i^{in}}, \widehat{s_i^{out}}) \leftarrow \widehat{\mathsf{GProg}}(1^\kappa, 1^{\log m}, 1^{t_i}, P_i)$, $\widehat{x}_i \leftarrow \widehat{\mathsf{GInput}}(1^\kappa, x_i, \widehat{s_i^{in}}, \widehat{s_i^{out}} - 1, \widehat{w})$. Finally, we consider the sequential execution of the garbled programs for $i = 1 \dots \ell$: $y'_i \leftarrow \widehat{\mathsf{GEval}}^{\widehat{D}_{i-1}}(\widehat{P}_i, \widehat{x}_i)$ which updates the garbled database to $\widehat{D}_i$.

Correctness. Our goal is to demonstrate that

$$\Pr[(y'_1, \dots, y'_\ell) = (y_1, \dots, y_\ell)] = 1.$$

Since $\widehat{\mathsf{GEval}}$ in our construction directly calls the underlying UMA-secure multi-program GRAM scheme for evaluation, the correctness of the underlying scheme guarantees that $(y'_1, \dots, y'_\ell) = (P_1^*(x_1, s^*), \dots, P_\ell^*(x_\ell, s^*))^{D^*}$. Then by the correctness of the ORAM scheme, $(P_1^*(x_1, s^*), \dots, P_\ell^*(x_\ell, s^*))^{D^*} = (P_1(x_1), \dots, P_\ell(x_\ell))^D = (y_1, \dots, y_\ell)$.

Security. For any programs $P_1, \dots, P_\ell$, database D, and inputs $x_1, \dots, x_\ell$, let

$$\text{REAL}^{D,\{P_i,x_i\}} = (\widehat{D}_0, \widehat{P}_i, \widehat{x}_{i}{}_{i=1}^{\ell})$$

Our goal is to construct a simulator Sim such that for all $D, \{P_i, x_i\}$, $\text{REAL}^{D,\{P_i,x_i\}} \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\kappa, 1^m, \{1^{t_i}, y_i\})$. We let OSim be the ORAM simulator, and USim be the simulator for the UMA-secure multi-program GRAM scheme. The procedure Sim proceeds as follows.

1. Compute $(D^*, \mathsf{MemAccess}) \leftarrow \mathsf{Sim}(1^\kappa, 1^m, \{1^{t_i}, y_i\}_{i=1}^\ell)$.
2. Compute $(\tilde{D}, \{\tilde{P}_i, \tilde{x}_i\}_{i=1}^\ell) \leftarrow \mathsf{Sim}(1^\kappa, 1^{m'}, \{1^{t'_i}, y_i\}_{i=1}^\ell, D^*, \mathsf{MemAccess})$, where m' is the size of D^* and t'_i is the running time of the oblivious program i.
3. Output $(\widehat{D}_0, \widehat{P}_i, \widehat{x}_{i}{}_{i=1}^{\ell}) = (\tilde{D}, \{\tilde{P}_i, \tilde{x}_i\}_{i=1}^\ell)$.

We now prove the output of the simulator is computationally indistinguishable from the real distribution. For any $D, \{P_i, x_i\}$, we define a series of hybrid distributions $\mathbf{Hyb}_0, \mathbf{Hyb}_1, \mathbf{Hyb}_2$ with $\mathbf{Hyb}_0 = \text{REAL}^{D,\{P_i,x_i\}}$, and $\mathbf{Hyb}_2 = \mathsf{Sim}(1^\kappa, 1^m, \{1^{t_i}, y_i\}_{i=1}^\ell)$, and argue that for $j = 0, 1$ we have $\mathbf{Hyb}_j \overset{\text{comp}}{\approx} \mathbf{Hyb}_{j+1}$.

- $\mathbf{Hyb}_0$: This is the real distribution $\text{REAL}^{D,\{P_i,x_i\}}$.
- $\mathbf{Hyb}_1$: Use the correctly generated (D^*, s^*) from $\widehat{\mathsf{GData}}$ and P_i^* from $\widehat{\mathsf{GProg}}$ and execute $(P_1^*(x_1, s^*), \dots, P_\ell^*(x_\ell, s^*))^{D^*}$ to obtain $\{y_i\}$ and a sequence of memory accesses $\mathsf{MemAccess}$. Run $(\tilde{D}, \{\tilde{P}_i, \tilde{x}_i\}_{i=1}^\ell) \leftarrow \mathsf{USim}(1^\kappa, 1^{m'}, \{1^{t'_i}, y_i\}_{i=1}^\ell, D^*, \mathsf{MemAccess})$ and output$(\widehat{D}_0, \widehat{P}_i, \widehat{x}_{i}{}_{i=1}^{\ell}) = (\tilde{D}, \{\tilde{P}_i, \tilde{x}_i\}_{i=1}^\ell)$.
- $\mathbf{Hyb}_2$: This is the simulated distribution $\mathsf{Sim}(1^\kappa, 1^m, \{1^{t_i}, y_i\}_{i=1}^\ell)$.

We now show that adjacent hybrid distributions are computationally indistinguishable.

$\mathbf{Hyb}_0 \overset{\text{comp}}{\approx} \mathbf{Hyb}_1$: Let $\mathcal{A}$ be a PPT distinguisher between these two distributions for some $D, \{P_i, x_i\}$. We construct an algorithm $\mathcal{B}$ that breaks the UMA-security of the underlying GRAM scheme that proceeds as follows. First, $\mathcal{B}$ runs $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D), P_i^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log m}, 1^{t_i}, P_i)$ and declares $D^*, \{P_i^*, x_i^+ = (x_i, s^*)\}$ as the challenge database, programs and inputs for the multi-program UMA-security GRAM game. The UMA-security challenger then outputs $(\tilde{D}', \{\tilde{P}'_i, \tilde{x}'_i\}_{i=1}^\ell)$ and $\mathcal{B}$ must output a

guess whether it is real or simulated. In order to do so, $\mathcal{B}$ sets $(\widehat{D}', \{\widehat{P}'_i, \widehat{x}'_i\}_{i=1}^{\ell}) = (\tilde{D}', \{\tilde{P}'_i, \tilde{x}'_i\}_{i=1}^{\ell})$ and forwards this as the challenge to $\mathcal{A}$. $\mathcal{B}$ then outputs the same guess as $\mathcal{A}$.

Observe that if the UMA challenger outputs the real values, then $(\widehat{D}', \{\widehat{P}'_i, \widehat{x}'_i\}_{i=1}^{\ell})$ is distributed identically as if it were generated from $\mathbf{Hyb}_0$, and if the UMA challenger outputs simulated values, then $(\widehat{D}', \{\widehat{P}'_i, \widehat{x}'_i\}_{i=1}^{\ell})$ is distributed identically as if it were generated from $\mathbf{Hyb}_1$. Therefore, $\mathcal{A}$ distinguishes with the same probability as $\mathcal{B}$, which is negligible by the multi-program UMA-security of the underlying GRAM scheme.

$\mathbf{Hyb}_1 \overset{\text{comp}}{\approx} \mathbf{Hyb}_2$: Let $\mathcal{A}$ be a PPT distinguisher between these two distributions for some $D, \{P_i, x_i\}$. We construct an algorithm $\mathcal{B}$ that breaks the security of the underlying ORAM scheme that proceeds as follows. First, $\mathcal{B}$ announces $D, \{P_i, x_i\}$ as the challenge database, programs, and inputs for the ORAM security game. The ORAM challenger then outputs $(D^{*\prime}, \mathsf{MemAccess}')$ which is either real or simulated. Then, $\mathcal{B}$ computes $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$ and runs $(\tilde{D}', \{\tilde{P}'_i, \tilde{x}'_i\}_{i=1}^{\ell}) \leftarrow \mathsf{USim}(1^\kappa, 1^{m'}, \{1^{t'_i}, y_i\}, D^{*\prime}, \mathsf{MemAccess}')$. Next, $\mathcal{B}$ sets $(\widehat{D}', \{\widehat{P}'_i, \widehat{x}'_i\}_{i=1}^{\ell}) = (\tilde{D}', \{\tilde{P}'_i, \tilde{x}'_i\}_{i=1}^{\ell})$ and forwards this to $\mathcal{A}$. $\mathcal{B}$ then outputs the same guess as $\mathcal{A}$.

Observe that if the ORAM challenger outputs the real values, then $(\widehat{D}', \{\widehat{P}'_i, \widehat{x}'_i\}_{i=1}^{\ell})$ is distributed identically as if it were generated from $\mathbf{Hyb}_1$, and if the ORAM challenger outputs simulated values, then $(\widehat{D}', \{\widehat{P}'_i, \widehat{x}'_i\}_{i=1}^{\ell})$ is distributed identically as if it were generated from $\mathbf{Hyb}_2$. Therefore, $\mathcal{A}$ distinguishes with the same probability as $\mathcal{B}$, which is negligible by the security of the underlying ORAM scheme.

□