2015 IEEE 56th Annual Symposium on Foundations of Computer Science

Black-Box Garbled RAM

Sanjam Garg
Computer Science Division
University of California, Berkeley
Berkeley, USA
Email: sanjamg@berkeley.edu

Steve Lu
Department of Computer Science
University of California, Los Angeles
Los Angeles, USA
Email: stevelu@cs.ucla.edu

Rafail Ostrovsky
Department of Computer Science
University of California, Los Angeles
Los Angeles, USA
Email: rafail@cs.ucla.edu

Abstract

Garbled RAM, introduced by Lu and Ostrovsky, enables the task of garbling a RAM (Random Access Machine) program directly, there by avoiding the inefficient process of first converting it into a circuit. Garbled RAM can be seen as a RAM analogue of Yao's garbled circuit construction, except that known realizations of Garbled RAM make non-black-box use of the underlying cryptographic primitives.

In this paper we remove this limitation and provide the first black-box construction of Garbled RAM with polylogarithmic overhead. Our scheme allows for garbling multiple RAM programs being executed on a persistent database and its security is based only on the existence of one-way functions. We also obtain the first secure RAM computation protocol that is both constant round and makes only black-box use of one-way functions in the Oblivious Transfer hybrid model.

Keywords

Garbled RAM; Black-Box Cryptography; One-Way Functions; Secure Computation

I. Introduction

Alice wants to store a large private database D on the cloud in an encrypted form. 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. Can we do this?

The above problem is a special case of the general problem of secure computation [Yao82], [GMW87]. In the past three decades of work, both theoretical and practical improvements have been pushing the limits of the overall efficiency of such schemes. However most of these constructions work only for circuits and securely computing a RAM program involves the inefficient process of first converting it into a circuit. For example, 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, in the example above, 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. Using fully homomorphic encryption [Gen09] we can reduce the size of Alice's message, but the cloud still needs to compute on 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. For example, in the case of binary search the size of the database can be exponentially larger than execution path of the insecure solution. In other words security comes at the cost of a exponential overhead. We note that additionally even in settings where the size of the database is small, generic transformations from Random Access Machine (RAM) programs with running time T result in a circuit of size $O(T^3 \log T)$ [CR73], [PF79], which can be prohibitively inefficient.

Secure computation for RAM programs: Motivated by the above considerations, various secure computation techniques that work directly for RAM programs have been developed. However all known results have interesting theoretical bottlenecks that influences efficiency, either in terms of *round complexity* or in their *non-black-box* use of cryptographic primitives.

- For instance, Ostrovsky and Shoup [OS97] show how general secure RAM computation can be done using oblivious RAM techniques [Gol87], [Ost90], [GO96]. Subsequently, Gordon et al. [GKK$^+$12] demonstrated an efficient realization based on specialized number-theoretic protocols. In follow up works, significant asymptotic and practical efficiency improvements have been obtained by Lu and Ostrovsky [LO13a] and by Wang et al. [WHC$^+$14]). However, all these works require round complexity on the order of the running time of the program.
- In another recent line of work [LO13b], [GHL$^+$14], [GLOS15], positive results on round efficient secure computation for RAM programs have been achieved. However all these constructions are inherently non-black-box in their use of cryptographic primitives.[1] These improvements are obtained by realizing the notion of garbled random-access

[1]We note that several other cutting-edge results [GKP$^+$13], [GHRW14], [CHJV14], [BGT14], [LP14] have been obtained in non-interactive secure computation over RAM programs but they all need to make non-black-box use of cryptographic primitives. Additionally some of these construction are based on strong computational assumptions such as [Reg05], [GGH$^+$13b], [GGH13a]. We skip discussing this further and refer the reader to [GLOS14, Appendix A] for more details.

0272-5428/15 $31.00 © 2015 IEEE
DOI 10.1109/FOCS.2015.22

IEEE computer society

machines (garbled RAMs) [LO13b] as a method to garble RAM programs directly, a RAM analogue of Yao's garbled circuits [Yao82].

In particular, we 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 $P^D(x)$.

Starting with Impagliazzo-Rudich [IR90], [IR89], researchers have been very interested in realizing cryptographic goals making just black-box use of underlying primitive. It has been the topic of many important recent works in cryptography [IKLP06], [PW09], [Wee10], [GLOV12], [GOSV14]. On the other hand, the problem of realizing black-box construction for various primitive is still open, e.g. multi-statement non-interactive zero-knowledge [BFM88], [FLS99], [GOS06] and oblivious transfer extension [Bea96].[2] From a complexity perspective, black-box constructions are very appealing as they often lead to conceptually simpler and qualitatively more efficient constructions.[3]

Motivated by low round complexity and black-box constructions, in this work, we ask if we can achieve the best of both worlds. In particular:

Can we construct garbled RAM programs with only polylogarithmic overhead making only black-box use of cryptographic primitives?

A. Our Results

In this paper, we provide the first construction of a fully black-box garbled RAM, i.e. both the construction and the security reduction make only black-box use of underlying cryptograhpic primitives (one-way functions). The security of our construction is based on the existence of one-way functions alone. We state this as our main theorem:

Main Theorem (Informal). *Assuming only the existence of one-way functions, there exists a secure* black-box *garbled RAM scheme, where the size of the garbled database is $\tilde{O}(|D|)$, size of the garbled input is $\tilde{O}(|x|)$ and the size of the garbled program and its evaluation time is $\tilde{O}(T)$ where T is the running time of program P. Here $\tilde{O}(\cdot)$ ignores* $\mathsf{poly}(\log T, \log |D|, \kappa)$ *factors where κ is the security parameter* [4]*. Since garbled RAM implies one-way functions, this can be stated as an equivalence of primitives.*

Just as in previous works on garbled RAM [LO13b], [GHL+14], [GLOS15], our construction allows for maintaining persistent database across execution of multiple programs on the garbled memory. Also as in [GKP+13], [LO13b], [GHL+14], [GLOS15], 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.

Secure RAM computation: We also obtain the first one-round secure computation protocol for RAM programs that makes only black-box use cryptography in the OT-hybrid model. A very unique feature of this construction is that it allows for asymmetric load in terms of storage costs, i.e., only one party stores the encrypted database. To the best of our knowledge no previous solutions allowed for an encrypted database based of private information of both parties to be stored on just one party, and yet allow secure RAM computation on it using black-box methods alone. This makes our constructions particularly relevant in the context of secure outsourced computation. Our garbled circuit generation algorithms all simply rely on a key for a pseudo-random function, and therefore can be also outsourced and generated by an external party holding the key in a manner similar to the work of Ananth et al. [ACG+14].

II. Our Techniques

We start by recalling briefly the high level idea behind the previous garbled RAM constructions [LO13b], [GHL+14], [GLOS15] and its follow up works. This serves as a good starting point in explaining the technical challenges that come up in realizing grabled RAM making only *black-box* use of cryptographic primitives.

[2] Interestingly for oblivious transfer extension we do know black-box construction based on stronger assumptions [IKNP03].

[3] Additionally, black-box constructions enable implementations agnostic to the implementation of the underlying primitives. This offers greater flexibility allowing for many optimizations, scalability, and choice of implementation.

[4] Although it is typically assumed that κ is polynomially related to M, one can redefine the security parameter to be as small as $\omega(1)\log^2 M$ and still efficiently achieve correctness and security that is all but negligible in M directly.

What makes black-box grabled RAM hard: We view 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 (that can grow dynamically at run-time though for simplicity we consider a D with M data elements). At a high level, known garbled RAM construction proceed in two steps. First a garbled RAM scheme is constructed under 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). Next this weaker security guarantee is amplified to get full security by using oblivious RAM. Both these steps introduce a non-black-box use of cryptographic primitives. Besides some technical details, the second step can actually be made black-box just by using statistical oblivious RAM [Ajt10], [DMN11], [SCSL11], [SvDS+13], [CLP14], though these statistical schemes do not protect the memory contents which will need to be addressed. Next we describe the challenges we need to overcome in realizing a black-box construction with UMA security.

At a very high level, known garbled RAM constructions with UMA security construct the garbled memory in the following way. 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 key for a pseudorandom function (PRF) F. 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 CPU step 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 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. Since the location to be read, i, is generated dynamically at run time the values $F_s(i, 0)$ and $F_s(i, 1)$ must be computed inside the garbled circuit. This is exactly where non-black-box use of the PRF is made.

More generally, there appears to be a fundamental barrier in this technique. Note that the data in the memory has to be stored in some encrypted form. In the above case it was stored as the output of a PRF evaluation. Whenever a bit is to be read from memory, it will need to be internally decrypted to recover what the value being read is, this makes the need for non-black-box use of cryptography essential. In fact, if we limit ourselves to black-box constructions then we do not know any garbled RAM solutions that are any better than the trivial solution of converting the RAM program to a circuit.

Our starting idea: dynamic memory: Our starting idea to recover from the above problem is to replace "static" memory made of various ciphertexts, as has been done in previous works; with a "dynamic" memory consisting of various garbled circuits. More specifically, in our new solution, we envision reading from "memory" in a new way. Unlike previous work, we envision reads from memory are achieved by passing the control flow of the program itself to the memory. We explain how garbled memory is activated: a small number of garbled circuits in the garbled memory are evaluated in sequence. These garbled circuits are connected in the sense that one garbled circuit will output a valid garbled input for the next garbled circuit by having both the zero and one labels hardwired within it. Eventually the control reaches back to the main program and additionally carries the read bit along with. Note that the actual garbled circuits that are fired inside the garbled memory are dynamically decided based on the location being read. Looking ahead, in our final construction they will also depend on the previous state of the garbled memory. We describe this later.

Next, we describe a plausible arrangement of garbled circuits in the garbled memory for realizing the above intuition. Let M be the size of memory. Imagine a tree of garbled circuits where the root garbled circuit has hardcoded in it the input labels for both its children. Similarly the left child garbled circuit has the input labels of its two children and so on. Finally the leaf garbled circuits, which are M in number, area such that each contains a bit of the database hardcoded in them. Our root garbled circuit takes as input two labels label^0 and label^1 and a location to be read. The root garbled circuit based on the location that needs to be read can activate its left or its right child garbled circuit, ultimately leading to the activation of a leaf garbled circuit which outputs either label^0 or label^1 based on whether the stored bit in it is 0 or 1. This enables a black-box way of reading a bit from memory.

However, the key challenge in realizing the above setup is that after only one read a sequence of garbled circuits from the root to a leaf in the garbled memory have been consumed/destroyed. Hence if we are to continue using the garbled memory further then we must provide "replacements" for the used garbled circuits. To get better insight into the issues involved, we start by describing a very natural dynamic replacement solution for this problem which actually fails because of rather subtle reasons. We believe that this solution highlights the technical issues involved.

Providing "replacements" dynamically: Our first attempted solution for overcoming the above challenge is to provide "generic" garbled circuits that can replace the specific garbled circuits that are used during a read. As mentioned earlier,

during a read, garbled circuits corresponding to a path from the root to a leaf are fired and in the process consumed. It is exactly these circuits that we need to replace. So corresponding to every read we could provide a sequence of garbled circuits to exactly replace these consumed garbled circuits. These replacement garbled circuits could be prepared to have the input labels of their new child already hardcoded in them, though some information needs to be provided at run-time.

Unfortunately this attempted approach has a very subtle bug — relating to a circularity in the parameter sizes. The problem is that the size of the additional inputs of the "replacements" provides the input labels for the "regular" input wires, but this information must be passed by the "regular" wires. In other words, if this scheme is to work, then we need to have the "replacement" garbled circuits be smaller than the garbled circuit that are being consumed which will leads to a blow up in the size of the first circuit. This appears to be a fundamental problem with this approach. We believe that black-box techniques cannot be used to fix this problem if only dynamic "replacements" are provided.

Providing "replacements" statically: Our second stab at the problem is to include for each node of the tree not just one garbled circuit but instead a sequence of garbled circuits. Of course we still need to respect the relationship that each garbled circuits needs to have the ability to activate its left and right child garbled circuits. Now that we have a sequence of garbled circuits for every node it is not clear which garbled circuits in its children sequences should a garbled circuit be connected with. A very simple strategy would be to have T garbled circuits in each sequence corresponding to each node, where T is the number of reads the garbled memory is designed for. We can connect the i^{th} garbled circuit in each sequence with the i^{th} garbled circuit in its children sequences. However, this leads to a garbled memory of size $T \cdot M$, something that is much larger than what we want, and defeating the purpose of this approach.

Note that even if we assumed that reads were *uniform* amongst the leaves, if we have a total of T reads happen at the root node then with a constant probability we expect a discrepancy of $\sqrt{T}$ between the minimum number and maximum number of reads that go left. This means that we now need a window of size $\sqrt{T}$ which is, which is still prohibitively large.

Defeating imbalances — having more circuits and fast-forwarding: Our main idea for dealing with the imbalances (which causes large window size) is to have more circuits in every sequence for each node. Indeed, these extra circuits serve two purposes. First, these extra circuits serve as a buffer in case we go beyond expectation. Secondly, when we are too far behind expectation, these extra circuits will be consumed *faster* to enforce that we are always within the window of keys that the parent node has. The key insight is that instead of having a fixed additive factor, the child pointers dynamically moves beyond the expectation (and enough standard deviations to achieve exopnentially small probability of failure) *relative* to the current location. As such, in earlier time steps there is less of a "stretch", whereas in later time steps there is more stretch. This resolves the tension between having too many stretch circuits yet still having enough to make sure you do not run out.

More specifically, our goal is to shrink the key window size down from $\sqrt{T}$ to a value that grows only with the security parameter. We shrink the window size using the following strategy: keep the window always well ahead of number of garbled circuits that could possibly be consumed (by the Chernoff bound), and provide a method to move into the window when lagging behind. In order to achieve this we introduce two new ideas, which when combined accomplish this strategy. First, each circuit has the option to "fast forward" or "burn" by passing on data to its *successor* circuit in the same node, specifically the next garbled circuit in the sequence. This "fast forwarding" is enough to ensure that the children garbled circuits always remain in the appropriate window. The second idea allows a parent garbled circuit to be able to evaluate old circuits that have fallen out of this window. Note that we only need the parent to be able to evaluate a *single* old circuit since whenever the child node is activated it will burn garbled circuits within its own sequence pushing it back into the window. Thus, we pass from circuit to circuit the keys to the first unused left and right child garbled circuits (and when consumed, will be replaced by the next key inside the window).

This causes a tension in the parameters: even though the number of circuits is roughly halving each time we go down, the factor by which we must push it back up by is also growing geometrically. Setting this growth rate to be even constant, say $c > 1$, is problematic. We run into difficulties since there will be $T\frac{1}{2}^i c^i$ circuits per node, and 2^i nodes per level, all the way up to $\log M$ — resulting in $TM^{\log c}$ circuits, which is polynomial overhead in the size of the dataset. It turns out that even a very slow growth rate suffices and allows up to get desired efficiency properties. With a careful analysis, it turns out this is both efficient and will only run out of garbled circuits with negligible probability. The exact details will become apparent in the construction and proof.

Getting provable security: Although the previous techniques achieve correctness and efficiency, it is not immediately obvious why it should be secure. Indeed, inputs keys for one garbled circuits are actually hardwired at multiple places, and if we do not carefully account for all of these locations, we could be running into circularity issues in our solution when using security of garbled circuits. To accommodate this, we will have a key hardwired only at the *first* place it was expected to be hardwired and "passed" dynamically to later circuits. Each garbled circuit will still maintain the same window of keys as were available to it earlier but now they are dynamically passed by it to its successor (the next garbled circuit in a sequence).

Moving forward, new keys are collected and the old keys will be dropped so that the total number of keys being passed remains small. Using this mechanism we can ensure that a garbled circuit can dynamically "drop" keys that correspond to a child garbled circuit whose security needs to be relied on in the proof. Based on this, in the hybrid argument, we argue that whenever a garbled circuit is replaced by a simulated version, we have all instances of its keys have been "dropped."

A technical issue also arises with the fact at the end some of the unevaluated garbled circuits remain. The problem lies in the fact that some of their input keys have also been revealed. We handle this issue by providing a generic transformation that ensures that garbled circuits are indistinguishable from noise as long as input keys for even one wire are not disclosed.

Final touches: In explaining the technical ideas above made several simplifying assumptions. We now provides some ideas on how to remove these limitations.

- **Arbitrary Memory Access.** As mentioned above, we can achieve a GRAM solution for programs with arbitrary access pattern by first compiling it with an ORAM that has uniform access pattern. Programs compiled with statistical ORAM do not actually have uniform memory access, but rather a *leveled* uniform access pattern, where the accesses in each level is uniformly distributed. We deal with this technicality by breaking our memory down into levels where access in each separate memory is uniform. Alternatively, we can *bias* the distribution where a leveled ORAM structure is flattened into memory: for example, we know that the a memory location corresponding to some tree node is accessed twice as often as its children, thus when we build our circuits we can incorporate and absorb this distribution into our scheme.
- **Replenishing.** Since we generate a fixed amount of garbled circuits for the garbled memory, this places a bound on the number of reads the memory can be used for. We observe that is we sent the number of reads to be equal the size of memory then this give us enough reads in parallel to with the entire memory can be replenished to allow for anther size of memory number of reads and so on. In our construction the garbled circuits are generated in a highly independent fashion and so more garbled circuits can be provided on the fly. Furthermore, this can be seamlessly amortized (the amortized overhead can be absorbed into the polylog factors) where the garbling algorithm for a T-time program can generate enough garbled circuits to support T more steps in the future. Finally, this strategy can also be used to accommodate memory that is dynamically growing.
- **Writing.** Writing in our construction is achieved in a way very similar to the reading. Reading in our scheme involves having a leaf garbled circuit pass on the value to the main circuit and simultaneously pass on the stored data value in it to its successor, so that it could be read again. During writing a garbled circuits passes the value to be written to its successor instead of the value previously stored.

A. Roadmap

We now lay out a roadmap for the remainder of the paper. In Section III, we give necessary background and definitions for the RAM model, garbled circuits, and garbled RAM. In Section IV we give the warmup heuristic construction of our result. We analyze the cost and correctness of the solution in Section V. We extend our construction to a secure one in Section VI and prove the security in Section VII (with the full proof in Section VIII).

III. 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].

A. 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.[5]

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.

[5]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.

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

$$C^P_{\mathsf{CPU}}(\mathsf{state}, \mathsf{data}) = (\mathsf{state}', \mathsf{R/W}, L, z)$$

This circuit takes as input the current CPU state state and a block "data". 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', a read or write bit $\mathsf{R/W}$, the next location to read/write $L \in [M]$, and a block z to write into the location ($z = \perp$ when reading). The sequence of locations and read/write values collectively form what is known as the *access pattern*, namely $\mathsf{MemAccess} = \{(L^\tau, \mathsf{R/W}^\tau, z^\tau, \mathsf{data}^\tau) : \tau = 1, \ldots, t\}$, and we can consider the weak access pattern $\mathsf{MemAccess2} = \{L^\tau : \tau = 1, \ldots, t\}$ of just the memory locations accessed.

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.

B. Garbled Circuits

Garbled circuits was first constructed by Yao [Yao82] (see e.g. 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.

- $\left(\tilde{C}\right) \leftarrow \mathsf{GCircuit}\left(1^\kappa, C, \{(w, b, \mathsf{lab}^w_b)\}_{w \in \mathsf{inp}(C), b \in \{0,1\}}\right)$: $\mathsf{GCircuit}$ takes as input a security parameter κ, a circuit C, and a set of labels lab^w_b for all the input wires $w \in \mathsf{inp}(C)$ and $b \in \{0,1\}$. This procedure outputs a *garbled circuit* $\tilde{C}$.
- It can be efficiently tested if a set of labels is meant for a garbled circuit.
- $y = \mathsf{Eval}(\tilde{C}, \{(w, \mathsf{lab}^w_{x_w})\}_{w \in \mathsf{inp}(C)})$: Given a garbled circuit $\tilde{C}$ and a garbled input represented as a sequence of input labels $\{(w, \mathsf{lab}^w_{x_w})\}_{w \in \mathsf{inp}(C)}$, Eval outputs an output y in the clear.

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:

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

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

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

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

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

C. 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 define both full security and a weaker variant known as Unprotected Memory Access 2 (UMA2) (similar to UMA security that appeared in [GHL$^+$14]), and we show how UMA2-secure Garbled RAM can be compiled with statistical Oblivious RAM to achieve full security.

Definition III.1. *A* (UMA2) secure single-program garbled RAM *scheme consists of four procedures* (GData, GProg, GInput, 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* $\tilde{D}$*.*
- $(\tilde{P}, s^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log M}, 1^t, P, s, m)$ *: Takes the description of a RAM program* P *with memory-size* M *as input. It also requires a key* s *and current time* m*. It then outputs a garbled program* $\tilde{P}$ *and an input-garbling-key* s^{in}*.*
- $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x, s^{in})$*: 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* $t \cdot \mathsf{poly}(\log M, \log T, \kappa)$*, 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* $M \cdot \mathsf{poly}(\log M, \log T, \kappa)$*, 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 $(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D)$*,* $(\tilde{P}, s^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log M}, 1^t, P, s, m)$*,* $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x, s^{in})$*.*

Security with Unprotected Memory Access (Full vs UMA2): For full or UMA2-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* MemAccess *we have that:*

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

where $(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D)$*,* $(\tilde{P}, s^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log M}, 1^t, P, s, m)$ *and* $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x, s^{in})$*, and* $y = P^D(x)$*. Note that unlike UMA security, the simulator does not have access to* D*. For full security, the simulator* Sim *does not get* MemAccess *as input.*

IV. The Construction

In this section we describe our construction for garbled RAM formally, namely the procedures (GData, GProg, GInput, GEval). In order to make the exposition simpler, in this section we will describe our construction making four simplifying assumptions, which will be all removed in our final construction.

1) **UMA2-security:** Here we will restrict ourselves to achieving UMA2-security alone Definition III.1 (UMA2). We note that this construction can then be amplified to get full security satisfying Definition III.1 (full) using a straightforward lemma (see the full version [GLO15]). This is essentially the transformation from previous works [LO13b], [GHL$^+$14], [GLOS15] except that we need to restrict ourselves to using statistical ORAMs [DMN11], [SCSL11], [SvDS$^+$13]. Note that this transformation is information theoretic and preserves the black-box nature of our construction.
2) **Uniform memory accesses:** We assume that the distribution of memory accesses of the programs being garbled are uniform. In the full version [GLO15], we also describe how this restriction can be removed and construction achieved even given any arbitrary probability distribution on memory reads. Essentially, we first compile our program with an Oblivious RAM that satisfies the property that the simulated access are uniformly distributed.
3) **First Step: Heuristic proof:** The construction described in this section is "heuristic" in the sense that we do not know how to prove its security. However we do not know of any concrete attacks against it. At a high level it suffers from a sort of a circular security problem. However this issue is rather easy to solve in our context. We describe the issue and the fix in Section VI to obtain a full security proof.

4) **Bounded reads:** We will describe our construction assuming that the total number of memory accesses (both read and write) made to the garbled memory is bounded by M, the size of the memory. In Section VIII, we explain how this restriction can be removed. In particular we will describe a memory replenishing mechanism for refilling the garbled memory as it used. This replenishing will involve some additional communication for each garbled program, while ensuring that the overhead of this replenishing information sent with each garbled program is small.

Notation: We use the notation $[n]$ to denote the set $\{0, \ldots, n-1\}$. For any string L, we use L_i to denote the i^{th} bit of L where $i \in [|x|]$ with the 0^{th} bit being the highest order bit. We let $L_{0\ldots j-1}$ denote the j high order bits of L. We will be using multiple garbled circuits and will need notation to represent bundles of input labels for garbled circuits succinctly. In particular, if $\mathsf{lab} = \{\mathsf{lab}^{i,b}\}_{i\in|x|,b\in\{0,1\}}$ describes the labels for input wires of a garbled circuit, then we let lab_x denote the labels corresponding to setting the input to x, i.e. the subset of labels $\{\mathsf{lab}^{i,x_i}\}_{i\in|x|}$. Similarly we will sometimes consider a collection of garbled circuits and denote the collection of labels for input wires of these garbled circuits with $\overline{\mathsf{lab}}$. Let i be an index of a garbled circuit in this collection then we let $\mathsf{lab}[i]_x$ denote the labels corresponding to setting the input to x of the i^{th} garbled circuit. Looking ahead, throughout our construction the inputs to the circuits we consider with be partitioned in into two parts, the red and the blue. We will use the colors red and blue to stress whether an input label corresponds to a red input wire or a blue input wire. We extend this coloring to collections of labels of the same color. We believe that this makes it much easier to read our paper and recommend reading it on a coloured screen or a colored printout.

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

We start by providing an informal description of the data garbling procedure, which turns out to be the most involved part of the construction. The formal description of GData is provided in Figure 4. Our garbled memory consists of two parts.

1) **Garbled Circuits:** Intuitively our garbled memory will be organized as a binary tree and each node of this tree will correspond to a sequence of garbled circuits. For any garbled circuit its *successor* is defined as the next garbled circuit in the sequence of garbled circuits corresponding to that node. Similarly we define *predecessor* as the previous garbled circuit in the sequence. For a garbled circuit all the garbled circuits in its parent node are referred to as its *parents*. Analogously we define *children*. These garbled circuits are obtained by fresh garblings of two separate circuits, one corresponding to the leaf nodes and the other corresponding to non-leaf nodes.
   For each of these garbled circuits, we will divide its input wires (and corresponding keys/lables) into two categories, the *r*ed input wires and the *b*lue input wires.
   Each garbled circuit will contain *all* input keys for its successor. Specifically this includes both the red and the blue input keys of its successor. Additionally each garbled circuit will contain a *subset* of input wires for a *subset* of its left and right children. Specifically, it will contain the blue input keys for a consecutive κ garbled circuits among its left children and a consecutive κ garbled circuits among its right children.
2) **Tabled garbled Information:** Additionally for each node in the tree as described above, the garbled memory consists of a table of information $\mathsf{Tab}(i, j)$, where (i, j) denotes a node in the tree.
   Looking ahead, as the memory is read from or written to these garbled circuits that constitute the garbled memory will actually be used. Furthermore if a garbled circuits corresponding to a node is being consumed then its predecessor must have previously already been consumed. The tabulated information will be the red input keys for the first unused garbled circuit for each node.

Circuits needed: Next we describe the two circuits, garblings of which will be used to generate the garbled memory. The circuits are described formally in Figures 2 and 3. The non-leaf node circuit takes as input some *recorded info* rec and a *query* q. Garbled labels for rec will be red and denoted as rKey and garbled labels for q will be blue and denoted as qKey. Although every single circuit will have its own unique rKey and qKey, when we refer to these in the context of some particular circuit, it will always refer to the keys of its successor and these values will be hard-coded in it. Additionally the circuit has hardcoded inside it its level i in the tree, its own position k within the sequence of garbled circuits at that node, garbled labels $\mathsf{rKey}, \mathsf{qKey}$ for its successor, and a collection of labels for the left and right child garbled circuits which we denote as $\overline{\mathsf{tKey}}$. Each tKey is a vector of exactly 2κ qKeys, the first κ correspond to qKeys for a contiguous block of κ of left child circuits (exactly which ones, we will describe later), and the last κ respectively correspond to labels for circuits in the right child.

The inputs are straightforward, rec contains indices to the first unused left and right child circuits as well as their qKeys. This allows us to either go left or right, although we will need to update the index and key as soon as we consume it, and we will replace it with something inside of $\overline{\mathsf{tKey}}$. The query q is simply a CPU query with one additional "goto" field $goto$ that informs where the first unused circuit in a node should be at to in order to fall inside the window of its parents $\overline{\mathsf{tKey}}$. If the current circuit $k < goto - 1$, we "burn" the circuit and pass our inputs onto our successor until it is precisely $goto - 1$,

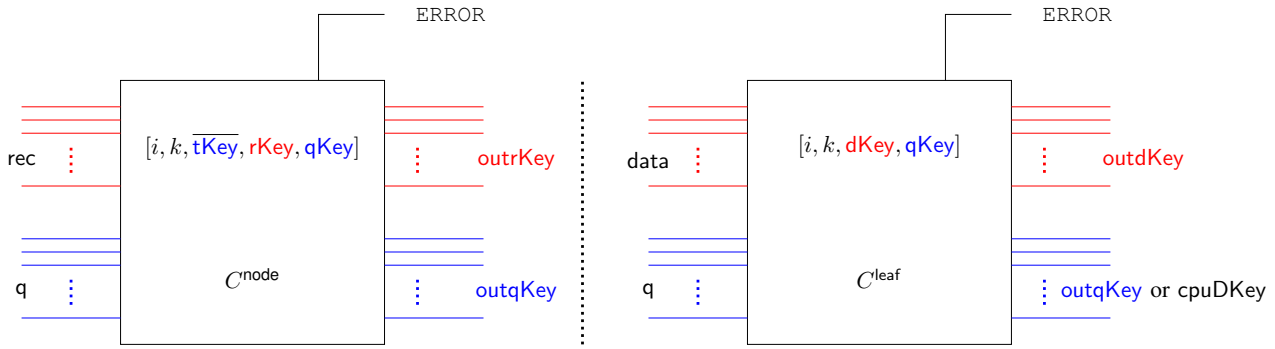

Figure 1. Memory circuits.

so that the first unused circuit is now indeed located at $goto$. In summary, we write $\mathsf{C}^{\mathsf{node}}[i, k, \overline{\mathsf{tKey}}, \mathsf{rKey}, \mathsf{qKey}](\mathsf{rec}, \mathsf{q})$ for non-leaf circuits.

$\mathsf{C}^{\mathsf{node}}[i, k, \overline{\mathsf{tKey}}, \mathsf{rKey}, \mathsf{qKey}]$
System parameters: ϵ (Will be set to $\frac{1}{\log M}$ as we will see later.)
Hardcoded parameters: $[i, k, \overline{\mathsf{tKey}}, \mathsf{rKey}, \mathsf{qKey}]$
Input: $(\mathsf{rec} = (\mathsf{lidx}, \mathsf{ridx}, \mathsf{oldLKey}, \mathsf{oldRKey}), \mathsf{q} = (goto, \mathsf{R/W}, L, z, \mathsf{cpuDKey}))$.

Set $\mathsf{p} := goto$ and $\mathsf{p}' := \left\lfloor \left(\frac{1}{2} + \epsilon\right) k \right\rfloor$. Set $\mathsf{lidx}' := \mathsf{lidx}$ and $\mathsf{ridx}' := \mathsf{ridx}$. Set $\mathsf{oldLKey}' := \mathsf{oldLKey}$ and $\mathsf{oldRKey}' := \mathsf{oldRKey}$. We now have three cases:

1) If $k < \mathsf{p} - 1$ then we output $(\mathsf{outrKey}, \mathsf{outqKey}) := (\mathsf{rKey}_{\mathsf{rec}'}, \mathsf{qKey}_{\mathsf{q}})$, where $\mathsf{rec}' := (\mathsf{lidx}', \mathsf{ridx}', \mathsf{oldLKey}', \mathsf{oldRKey}')$.
2) If $k \geq \mathsf{p} + \kappa$ then abort with output `OVERCONSUMPTION-ERROR-I`.
3) If $\mathsf{p} - 1 \leq k < \mathsf{p} + \kappa$ then:
   a) If $L_i = 0$ then,
      i) If $\mathsf{lidx} < \mathsf{p}'$ then set $\mathsf{lidx}' := \mathsf{p}', goto' := \mathsf{p}'$ and $\mathsf{oldLKey}' := \mathsf{tKey}[0]$. Else set $\mathsf{lidx}' := \mathsf{lidx} + 1, goto' := \mathsf{lidx}'$ and if $\mathsf{lidx}' < \mathsf{p}' + \kappa$ then set $\mathsf{oldLKey}' := \mathsf{tKey}[\mathsf{lidx}' - \mathsf{p}']$ else abort with `OVERCONSUMPTION-ERROR-II`.
      ii) Set $\mathsf{outqKey} := \mathsf{oldLKey}_{\mathsf{q}'}$, where $\mathsf{q}' := \mathsf{q}$ but with $goto'$ replacing $goto$.

      else
      i) If $\mathsf{ridx} < \mathsf{p}'$ then set $\mathsf{ridx}' := \mathsf{p}', goto' := \mathsf{p}'$ and $\mathsf{oldRKey}' := \mathsf{tKey}[\kappa]$. Else set $\mathsf{ridx}' := \mathsf{ridx} + 1, goto' := \mathsf{ridx}'$ and if $\mathsf{ridx}' < \mathsf{p}' + \kappa$ then set $\mathsf{oldRKey}' := \mathsf{tKey}[\kappa + \mathsf{ridx}' - \mathsf{p}']$ else abort with `OVERCONSUMPTION-ERROR-II`.
      ii) Set $\mathsf{outqKey} := \mathsf{oldRKey}_{\mathsf{q}'}$, where $\mathsf{q}' := \mathsf{q}$ but with $goto'$ replacing $goto$.
   b) Set $\mathsf{outrKey} := \mathsf{rKey}_{\mathsf{rec}'}$ where $\mathsf{rec}' := (\mathsf{lidx}', \mathsf{ridx}', \mathsf{oldLKey}', \mathsf{oldRKey}')$ and output $(\mathsf{outrKey}, \mathsf{outqKey})$.

Figure 2. Formal description of the nonleaf Memory Circuit.

Similarly, leaf circuits takes as input some *memory data* data and a *query* q. Here, the red key is dKey which corresponds to the garbled labels of data. A leaf circuit will have hardcoded inside it the current level $i = d$ in the tree, its own position k within the sequence of garbled circuits at that leaf, garbled labels $\mathsf{dKey}, \mathsf{qKey}$ for its successor. Since a leaf node has no further children, there is no need for $\overline{\mathsf{tKey}}$. We write $\mathsf{C}^{\mathsf{leaf}}[i, k, \mathsf{dKey}, \mathsf{qKey}](\mathsf{data}, \mathsf{q})$ for these circuits.

Actual data garbing: At a high level, we generate our garbled memory by garbling multiple instances of circuits described in Figures 2 and 3. The formal construction is provided in Figure 4. As mentioned earlier, these garbled circuits actually correspond to the nodes of a tree. Specifically, if the size of the database is $M = 2^d$, then the root node will contain roughly $M\kappa$ circuits, each nodes in subsequent level will contain roughly half that amount. More specifically, any node at level i contains at most $K_i = \left\lfloor \left(\frac{1}{2} + \epsilon\right)^i (M + i\kappa) \right\rfloor + \kappa$ garbled circuits. In total, the garbled memory will consist of $\sum_{i=0}^{d} (1 + 2\epsilon)^i (M + i\kappa) + \kappa$ garbled circuits. Looking ahead ϵ will be set to $\frac{1}{d}$ so that the this number is linear in $M + d\kappa$.

$\mathsf{C}^{\mathsf{leaf}}[i, k, \mathsf{dKey}, \mathsf{qKey}]$
System parameters: ϵ (Will be set to $\frac{1}{\log M}$ as we will see later.)
Hardcoded parameters: $[i, k, \mathsf{dKey}, \mathsf{qKey}]$
Input: $(\mathsf{data}, \mathsf{q} = (goto, \mathsf{R/W}, L, z, \mathsf{cpuDKey}))$.

Set $\mathsf{p} := goto$ and $\mathsf{p}' := \left\lfloor \left(\frac{1}{2} + \epsilon\right) k \right\rfloor$. We now have three cases:

1) If $k < \mathsf{p} - 1$ then we output $(\mathsf{outdKey}, \mathsf{outqKey}) := (\mathsf{dKey}_{\mathsf{data}}, \mathsf{qKey}_{\mathsf{q}})$.
2) If $k \geq \mathsf{p} + \kappa$ then abort with output `OVERCONSUMPTION-ERROR-I`.
3) If $\mathsf{p} - 1 \leq k < \mathsf{p} + \kappa$ then:
   a) If $\mathsf{R/W} = \mathsf{read}$ then output $(\mathsf{dKey}_{\mathsf{data}}, \mathsf{cpuDKey}_{\mathsf{data}})$, else if $\mathsf{R/W} = \mathsf{write}$ then output $(\mathsf{dKey}_z, \mathsf{cpuDKey}_z)$.

Figure 3. Formal description of the leaf Memory Circuit.

The algorithm $\mathsf{GData}(1^\kappa, D)$ proceeds as follows. Without loss of generality we assume that $M = 2^d$ (where $M = |D|$) where d is a positive integer. We calculate $\epsilon = \frac{1}{\log M}$. We set $K_0 = M$, and for each $i \in [d+1]$ and set $K_i = \left\lfloor \left(\frac{1}{2} + \epsilon\right) K_{i-1} \right\rfloor + \kappa$.

1) Let $s \leftarrow \{0,1\}^\kappa$.
2) Any $\mathsf{dKey}^{d,j,k}$ needed in the computation below is obtained as $F_s(\mathtt{data}||d||j||k)$. Similarly for any i, j, k, $\mathsf{rKey}^{i,j,k} := F_s(\mathtt{rec}||i||j||k)$ and $\mathsf{qKey}^{i,j,k} := F_s(\mathtt{query}||i||j||k)$. Finally,
$$\overline{\mathsf{tKey}}^{i,j,k} := \left\{ \underbrace{\left\{\mathsf{qKey}^{i+1,2j,\left\lfloor\left(\frac{1}{2}+\epsilon\right)k\right\rfloor+l}\right\}_{l\in[\kappa]}}_{\text{left}}, \underbrace{\left\{\mathsf{qKey}^{i+1,2j+1,\left\lfloor\left(\frac{1}{2}+\epsilon\right)k\right\rfloor+l}\right\}_{l\in[\kappa]}}_{\text{right}} \right\}.$$
3) For all $j \in [2^d], k \in [K_d]$,
$\tilde{C}^{d,j,k} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{leaf}}\left[d, k, \mathsf{dKey}^{d,j,k+1}, \mathsf{qKey}^{d,j,k+1}\right], \mathsf{dKey}^{d,j,k}, \mathsf{qKey}^{d,j,k}\right)$.
4) For all $i \in [d], j \in [2^i], k \in [K_i]$,
$\tilde{C}^{i,j,k} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{node}}\left[i, k, \overline{\mathsf{tKey}}^{i,j,k}, \mathsf{rKey}^{i,j,k+1}, \mathsf{qKey}^{i,j,k+1}\right], \mathsf{rKey}^{i,j,k}, \mathsf{qKey}^{i,j,k}\right)$.
5) For all $j \in [2^d]$, set $\mathsf{Tab}(d, j) = \mathsf{dKey}^{d,j,0}_{D[j]}$.
6) For all $i \in [d], j \in [2^i]$, set $\mathsf{Tab}(i, j) := \mathsf{rKey}^{i,j,0}_{\mathsf{rec}^{i,j,0}}$, where $\mathsf{rec}^{i,j,0} := (0, 0, \mathsf{qKey}^{i+1,2j,0}, \mathsf{qKey}^{i+1,2j+1,0})$
7) Output $\tilde{D} := \left(\left\{\tilde{C}^{i,j,k}\right\}_{i\in[d+1], j\in[2^i], k\in[K_i]}, \{\mathsf{Tab}(i,j)\}_{i\in[d+1], j\in[2^i]}\right)$ and s.

Figure 4. Formal description of GData.

In order to simplify generation of garbled circuits, we generate all the labels needed for generation of these garbled circuits as the outputs of a PRF on appropriate input values under a fixed seed s. Looking ahead, this will be crucial in extending our construction to allow for generating memory replenishing information. This is elaborated upon in Section VIII.

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

We start by defining a sub-circuit that will be needed in describing the program garbling in Figure 5. This circuit basically performs one step of the CPU and provides input lables for a root garbled circuit in the garbled memory. The formal description of program garbling itself in provided in Figure 6. The garbled program is obtained by garbling multiple CPU step circuits where very rough each circuit provides the input labels for the next CPU step and for the root circuit of the garbled memory, which then enables reading data from memory.

$\mathsf{C}^{\mathsf{step}}[t, \mathsf{rootqKey}, \mathsf{cpuSKey}, \mathsf{cpuDKey}]$
Hardcoded parameters: $[t, \mathsf{rootqKey}, \mathsf{cpuSKey}, \mathsf{cpuDKey}]$
Input: $(\mathsf{state}, \mathsf{data})$.

Compute $(\mathsf{state}', \mathsf{R/W}, L, z) := C^P_{\mathsf{CPU}}(\mathsf{state}, \mathsf{data})$. Set $\mathsf{q} := (goto = t+1, \mathsf{R/W}, L, z, \mathsf{cpuDKey})$ and output $\mathsf{rootqKey}_{\mathsf{q}}$ and $\mathsf{cpuSKey}_{\mathsf{state}'}$, unless we are halting in which case only output state' in the clear.

Figure 5. Formal description of the step circuit.

The $\mathsf{GProg}(1^\kappa, 1^{\log M}, 1^t, P, s, m)$ procedure proceeds as follows.

1) Any $\mathsf{cpuSKey}^\tau$ needed in the computation below is obtained as $F_s(\texttt{CPUstate}||\tau)$, and any $\mathsf{cpuDKey}^\tau$ is obtained as $F_s(\texttt{CPUdata}||\tau)$.
2) For $\tau = m, \ldots, m+t-1$ do:
   a) Set $\mathsf{qKey}^{0,0,\tau} := F_s(\texttt{query}||0||0||\tau)$.
   b) $\tilde{C}^\tau \leftarrow \mathsf{GCircuit}\left(1^\kappa, C^{\mathsf{step}}\left[\tau, \mathsf{qKey}^{0,0,\tau}, \mathsf{cpuSKey}^{\tau+1}, \mathsf{cpuDKey}^{\tau+1}\right], \mathsf{cpuSKey}^\tau, \mathsf{cpuDKey}^\tau\right)$
3) Output $\tilde{P} := \left(m, \{\tilde{C}^\tau\}_{\tau\in\{m,\ldots,m+t-1\}}, \mathsf{cpuDKey}_0^m\right)$, $s^{in} = \mathsf{cpuSKey}^m$

Figure 6. Formal description of GProg.

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

Informally, the GInput algorithm uses x 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 $\mathsf{GInput}(1^\kappa, x, s^{in})$ proceeds as follows.

1) Parse s^{in} as cpuSKey and output $\tilde{x} := \mathsf{cpuSKey}_x$.

Figure 7. Formal description of GInput.

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

The GEval procedure gets as input the garbled program $\tilde{P} = \left(m, \{\tilde{C}^\tau\}_{\tau\in\{m,\ldots,m+t-1\}}, \mathsf{cpuDKey}\right)$, the garbled input $\tilde{x} = \mathsf{cpuSKey}$ and random access into the garbled database $\tilde{D} = (\{\tilde{C}^{i,j,k}\}_{i\in[d+1],j\in[2^i],k\in[K_i]}, \{\mathsf{Tab}(i,j)\}_{i\in[d+1],j\in[2^i]})$. Intuitively the GEval is very simple. It proceeds by executing a subset of the garbled circuits from the garbled program and the garbled memory in a specific order which is decided dynamically based on the computation. 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 for at most two distinct circuits. Among these two circuits, only one is such that all its input labels have been provided, and this circuit is executed next. The unused input labels are stored in memory table Tab to be used at a later point. Next we provide the formal description of GEval in Figure 8.

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

1) Parse $\tilde{P}$ as $\left(m, \{\tilde{C}^\tau\}_{\tau\in\{m,\ldots,m+t-1\}}, \mathsf{cpuDKey}\right)$, $\tilde{x}$ as cpuSKey and $\tilde{D}$ as $\left(\left\{\tilde{C}^{i,j,k}\right\}_{i\in[d+1],j\in[2^i],k\in[K_i]}, \{\mathsf{Tab}(i,j)\}_{i\in[d+1],j\in[2^i]}\right)$.
2) For $\tau \in \{m, \ldots, m+t-1\}$ do:
   a) Evaluate $(\mathsf{cpuSKey}, \mathsf{qKey}) := \mathsf{Eval}(\tilde{C}^\tau, (\mathsf{cpuSKey}, \mathsf{cpuDKey}))$. If an output y is produced by Eval instead, then output y and halt.
   b) Set $i = 0, j = 0, k = \tau$.
   c) Evaluate $\mathsf{outputKey} := \mathsf{Eval}(\tilde{C}^{i,j,k}, (\mathsf{Tab}(i,j), \mathsf{qKey}))$.
      i) If outputKey is parsed as $(\mathsf{rKey}, \mathsf{qKey}^{i',j',k'})$ for some i', j', k', then set $\mathsf{Tab}(i,j) := \mathsf{rKey}, \mathsf{qKey} := \mathsf{qKey}^{i',j',k'}$, $(i,j,k) = (i',j',k')$ and go to Step 2c.
      ii) Otherwise, set $(\mathsf{dKey}, \mathsf{cpuDKey}) := \mathsf{outputKey}$,and $\mathsf{Tab}(i,j) := \mathsf{dKey}$ and $\tau := \tau + 1$.

Figure 8. Formal description of GEval.

V. Cost and Correctness Analysis

A. Overall Cost

Before we analyze the cost of the main algorithms, we first calculate the sizes of all the constituent variables and circuits. The database D has size $|D| = M$ elements, and each data element is of $|\mathsf{data}| = B$ bits. Garbled labels for each bit of an input wire are λ bits long. The complete garbled labels for n input bits takes up $2\lambda n$ bits. Furthermore, the current time step m or τ we upper bound by the total combined running time T. Of course, B, λ, T are all $\mathsf{poly}(\kappa)$, and for the two former values we simply absorb them into the $\mathsf{poly}(\kappa)$ term, whereas we keep T as a separate parameter for later use.

From this, we can compute $|\mathsf{cpuSKey}| = 2\lambda|\mathsf{state}|$ and $|\mathsf{cpuDKey}| = 2\lambda B$. A query q has size $|goto| + |\mathsf{R/W}| + |L| + |z| + |\mathsf{cpuDKey}| \leq \log T + 1 + \log M + B + 2\lambda B$, and $|\mathsf{qKey}| = 2\lambda|\mathsf{q}|$. Since dKey are just the labels for memory data, $|\mathsf{dKey}| = 2\lambda B$. Next, we compute the size of rec. Observe that $\mathsf{oldLKey}$ and $\mathsf{oldRKey}$ are simply qKeys, so we have $|\mathsf{rec}| = |\mathsf{lidx}| + |\mathsf{ridx}| + |\mathsf{oldLKey}| + |\mathsf{oldRKey}| \leq 2(\log T + |\mathsf{qKey}|)$. Finally, $\overline{\mathsf{tKey}}$ consist of 2κ qKeys and therefore have size $|\overline{\mathsf{tKey}}| = 2\kappa|\mathsf{qKey}|$.

Now we calcluate $|C^{\mathsf{node}}|$. Observe that the calculations within the circuit are primarily comparisons, and overall is at most polynomial in the size of the input and hardwired values. Thus $|C^{\mathsf{node}}| = \mathsf{poly}(|i|+|k|+|\overline{\mathsf{tKey}}|+|\mathsf{rKey}|+|\mathsf{qKey}|+|\mathsf{rec}|+|\mathsf{q}|) = \mathsf{poly}(\log M, \log T, \kappa)$ and so is its garbled version.

Next, we calcluate $|C^{\mathsf{leaf}}|$. We have $|\mathsf{C}^{\mathsf{leaf}}[i, k, \mathsf{dKey}, \mathsf{qKey}](\mathsf{data}, \mathsf{q})| = \mathsf{poly}(|i| + |k| + |\mathsf{dKey}| + |\mathsf{qKey}| + |\mathsf{data}| + |\mathsf{q}|) = \mathsf{poly}(\log M, \log T, \kappa)$ and so is its garbled version.

Finally, we calcluate $|C^{\mathsf{step}}|$. We assume that the plain CPU has size $\mathsf{poly}(\log M, \log T, \kappa)$. Since the step circuit simply computes the CPU circuit and does a few selections, we have: $|C^{\mathsf{step}}| = \mathsf{poly}(\log T + |\mathsf{qKey}| + |\mathsf{cpuSKey}| + |\mathsf{cpuDKey}| + |\mathsf{state}| + |\mathsf{data}| + |CPU|) = \mathsf{poly}(\log M, \log T, \kappa)$.

We can now calculate the cost of the individual algorithms.

1) Cost of GData*:* The algorithm $\mathsf{GData}(1^\kappa, D)$ first computes $O(M)$ $\mathsf{dKey}, \mathsf{rKey}, \mathsf{qKey}$ values, which only takes $M \cdot \mathsf{poly}(\log M, \log T, \kappa)$ steps. For each node at level $i < d$, it computes K_i garbled C^{node} circuits and tabulates an an rKey for each of the 2^i nodes. At level $i = d$, it computes K_d garbled C^{leaf} circuits and tabulates $M = 2^d$ dKeys. The output is of size equal to all the garbled circuits plus the size of the tabulated values plus one PRF key. Let $c = e^2$ and let $\epsilon = \frac{1}{\log M}$.

First, we show how to bound bound $K_i \leq \left(\frac{1}{2} + \epsilon\right)^i M + \sum_{j=0}^{i-1} \left(\frac{1}{2} + \epsilon\right)^j \kappa$. This can be shown by induction: $K_0 = M$, and by induction, $K_{i+1} = \left\lfloor \left(\frac{1}{2} + \epsilon\right) K_i \right\rfloor + \kappa \leq \left(\frac{1}{2} + \epsilon\right) K_i + \kappa \leq \left(\frac{1}{2} + \epsilon\right) \cdot \left[\left(\frac{1}{2} + \epsilon\right)^i M + \sum_{j=0}^{i-1} \left(\frac{1}{2} + \epsilon\right)^j \kappa\right] + \kappa \leq \left(\frac{1}{2} + \epsilon\right)^{i+1} M + \sum_{j=0}^{i+1-1} \left(\frac{1}{2} + \epsilon\right)^j \kappa$.

This bound can then be simplified to $K_i \leq \left(\frac{1}{2} + \epsilon\right)^i (M + i\kappa)$.

Thus, overall, we calculate the number of garblings of C^{node} as

$$\sum_{i=0}^{d-1} 2^i \cdot K_i \leq \sum_{i=0}^{d-1} 2^i \cdot \left(\frac{1}{2} + \epsilon\right)^i (M + i\kappa) \leq \sum_{i=0}^{d-1} (1 + 2\epsilon)^i (M + d\kappa) \leq \frac{(1 + 2\epsilon)^d - 1}{(1 + 2\epsilon) - 1} (M + d\kappa) \leq \frac{e^{2\epsilon d} - 1}{2\epsilon} (M + d\kappa)$$

Since $\epsilon = 1/d$, and garbling such circuits takes $\mathsf{poly}(\log M, \log T, \kappa)$ time, this overall takes $M \cdot \mathsf{poly}(\log M, \log T, \kappa)$ time and space. At the leaf level, it performs at most $2^d \cdot (\frac{1}{2} + \epsilon)^d (M + d\kappa)$ garblings of C^{leaf}. Again, this takes $\mathsf{poly}(\log M, \log T, \kappa) \cdot M$ time and space. Finally, there are $O(M)$ of rKey and dKey values stored in $\mathsf{Tab}(i, j)$, which is again $\mathsf{poly}(\log M, \log T, \kappa) \cdot M$.

2) Cost of GProg *:* The algorithm $\mathsf{GProg}(1^\kappa, 1^{\log M}, 1^t, P, s, m)$ computes t $\mathsf{cpuSKey}$s,$\mathsf{cpuDKey}$s, and qKeys. It also garbles t C^{step} circuits and outputs them, along with a single $\mathsf{cpuSKey}$. Since each individual operation is $\mathsf{poly}(\log M, \log T, \kappa)$, the overall time and space cost is $\mathsf{poly}(\log M, \log T, \kappa) \cdot t$.

3) Cost of GInput *:* The algorithm $\mathsf{GInput}(1^\kappa, x, s^{in})$ selects labels of the state key based on the state as input. As such, the time and space cost is $|\mathsf{cpuSKey}|$.

4) Cost of GEval *:* We first assume that an error does not occur in GEval. As we shall see in Section V-B that this occurs with all but negligible probability. We analyze how many circuits were consumed after T steps in order to obtain the amortized cost of GEval. We let k_i denote the maximum number of circuits consumed in some node at level i. At the root, exactly T circuits were consumed so $k_0 = T$, and in order for level i to not overflow, it must not have consumed more than $\left\lfloor \left(\frac{1}{2} + \epsilon\right) k_{i-1} \right\rfloor + \kappa$ circuits. By the same analysis of the bound of K_i, it must be the case that $k_i \leq \left(\frac{1}{2} + \epsilon\right)^i (T + i\kappa)$. Then no more than $\sum_{i=0}^{d} 2^i k_i$ circuits could have been consumed, each of which has evaluation cost at most $\mathsf{poly}(\log M, \log T, \kappa)$. It turns out this bound is slightly insufficient, and this is due to the case when $T < M$, the 2^i term is an overestimate. Indeed, if there are only T accesses, then there can be at most T nodes that were ever touched at a level. Using $min(2^i, T)$ as the bound on the number of nodes ever touched per level suffices:

$$\sum_{i=0}^{d} min(2^i, T) k_i \leq \sum_{i=0}^{d} min(2^i, T) \left(\frac{1}{2} + \epsilon\right)^i (T + i\kappa) \leq T \left(\sum_{i=0}^{d} min(2^i, T) \left(\frac{1}{2} + \epsilon\right)^i + \frac{min(2^i, T) i\kappa}{T}\right)$$

$$\leq T \left(\sum_{i=0}^{d} (2^i)(\left(\frac{1}{2} + \epsilon\right)^i) + \frac{(T) i\kappa}{T}\right) \leq T \left(d((1 + 2\epsilon)^d + d\kappa)\right) \leq T \left(d(e^2 + d\kappa)\right).$$

When accounting for the cost of each of these circuits being evaluated, this means that the amortized cost is $T \cdot \mathsf{poly}(\log M, \log T, \kappa)$ overall.

B. Correctness

Observe that as long as the memory data is correctly stored and passed on to the CPU step circuits, the scheme is correct. The only way this can fail to happen is if a query q fails to make it from the root to the leaf. In order to demonstrate this, we need to analyze two things. We must show that a parent ciruit will always output the proper qKey for the first unused child circuit, and we also must show that the errors `OVERCONSUMPTION-ERROR-I` and `OVERCONSUMPTION-ERROR-II` do not occur except with a negligible probability.

Lemma V.1. *Within C^{node},* lidx *always points to the first unused left child circuit which has* qKey *equal to* oldLKey*, and* ridx *always opint to the first unused right child circuit which has* qKey *equal to* oldRKey*.*

Proof: WLOG we show this for the left child. We prove this by induction on the current CPU step. In the base case, this is true due to the way GData set up the keys. Now suppose we are consuming some parent circuit and it was true for the previous circuit, i.e. lidx and oldLKey correctly point to the first unused left child circuit. Then it remains to show that lidx$'$ points to what will be the first unused left child circuit during the next CPU step, and that the updated old key oldLKey$'$ points to it. Recall $\mathsf{p}' = \lfloor(\frac{1}{2}+\epsilon)k\rfloor$, and by definition of GData, this is precisely the child circuit that tKey$[0]$ is the qKey of. If lidx $<$ p$'$ then the child circuit will burn until the $goto'$ circuit, which is exactly what lidx$'$ is set to be, and oldLKey$'$ is set to tKey$[0]$ which is precisely what lidx$' = goto'$ is set to. On the other hand, if lidx $> goto'$ then by definition, $goto' =$ lidx$' =$ lidx $+ 1$ and oldLKey$'$ holds the key for precisely the next circuit past lidx. But we know that the child node will consume exactly one circuit since $goto'$ is precisely one past lidx which by induction is the current child index, so lidx$'$ will point to the first unused child circuit and oldLKey$'$ is its key. ■

Lemma V.2. *The errors `OVERCONSUMPTION-ERROR-I` and `OVERCONSUMPTION-ERROR-II` do not occur except with a negligible probability.*

Proof: Again, WLOG we show this for the left child. Note that an error can never occur at the root, and that the error `OVERCONSUMPTION-ERROR-I` would occur if and only if an `OVERCONSUMPTION-ERROR-II` would have occured just before it. Thus, we bound the probability that an `OVERCONSUMPTION-ERROR-I` could occur. Suppose an error first occurs at some node (i, j) at the m_1-th circuit in this node. Then this means that the child lidx$'$ has become greater than $\mathsf{p}' + \kappa = \lfloor(\frac{1}{2}+\epsilon)m_1\rfloor + \kappa$. Since each time the left child is visited, many child circuits may be consumed due to burning, it might be difficult to figure out exactly how many child circuits were consumed if m_1 parent circuits were consumed. However, we can define a *synchronize* event, which is namely that the parent is on circuit k and the child is on circuit $\lfloor(\frac{1}{2}+\epsilon)k\rfloor$, or more precisely, when $goto' = \lfloor(\frac{1}{2}+\epsilon)k\rfloor$. We let $m_0 < m_1$ be the latest point for the parent for which this synchronize occured. We know that such an m_0 exists, since time $m_0 = 0$ is a valid solution.

Because there have been no more burns since that time, each time the left child was visited, exactly one circuit was also consumed. At m_0, exactly $\lfloor(\frac{1}{2}+\epsilon)m_0\rfloor$ child circuits were consumed, and at m_1, more than $\lfloor(\frac{1}{2}+\epsilon)m_1\rfloor + \kappa$ child circuits would have been consumed (if we did not break on error). During this time, $m_1 - m_0$ parent circuits were consumed, so the parent node was visited at most $m_1 - m_0$ times (it could be less due to burning), and we expect the child node to be visited $\mu = \frac{m_1-m_0}{2}$ times. For $t = 0, \ldots, m_1 - m_0$, let X_t denote the 0/1 random variable indicating that on time step $m_0 + t$ the left node was visited, and let $X = \sum_{t=0}^{m_1-m_0} X_t$. We calculate the probability that $Pr[X > \lfloor(\frac{1}{2}+\epsilon)m_1\rfloor + \kappa - \lfloor(\frac{1}{2}+\epsilon)m_0\rfloor]$ which is the probability that this error would have occurred. This becomes $Pr[X > \left(\left(\frac{1}{2}+\epsilon\right)(m_1 - m_0)\right) + \kappa - 1]$. Note that we can trivially condition on the case where $m_1 - m_0 > \kappa$, because otherwise $X < \kappa$ with probability 1, so we can conclude $\mu > \kappa/2$.

Substituting $\delta = 2\epsilon + \frac{\kappa-1}{\mu}$, this becomes $Pr[X > (1+\delta)\mu]$. Then by the Chernoff bound for $\delta > 0$, $Pr[X > (1+\delta)\mu] \leq \exp((\delta - (1+\delta)\ln(1+\delta))\mu)$.

Then reorganizing terms and using a second-order log approximation:

$$
\begin{aligned}
Pr[X > (1+\delta)\mu] &\leq \exp\left[\left(\delta + (1+\delta)\log\left(1 - \frac{\delta}{1+\delta}\right)\right)\mu\right] \leq \exp\left[\left(\delta + (1+\delta)\left(-\frac{\delta}{1+\delta} - \frac{\delta^2}{(1+\delta)^2}\right)\right)\mu\right] \\
&\leq \exp\left[-\frac{\delta^2\mu}{1+\delta}\right] \leq \exp\left[-\delta\mu\left(\frac{\delta}{1+\delta}\right)\right] \leq \exp\left[-(2\epsilon\mu + \kappa - 1)\left(\frac{2\epsilon + (\kappa-1)/\mu}{1 + 2\epsilon + (\kappa-1)/\mu}\right)\right] \\
&\leq \exp\left[-(2\epsilon\mu + \kappa - 1)\left(\frac{2\epsilon}{1+2\epsilon}\right)\right] \leq \exp\left[-(2\epsilon\mu + \kappa - 1)\,(\epsilon)\right] \leq \exp\left[-(2\epsilon^2\mu + \epsilon(\kappa - 1))\right]
\end{aligned}
$$

Since $\epsilon = \frac{1}{\log M}$, this is negligible.

■

VI. Secure Main Construction

We first provide the intuition of how we would like the proof would go through. Our goal is to construct a simulator Sim such that only given the access pattern and output (and not the database contents) it can simulate the view of the evaluator. The first observation is that the only point of the PRF F was to allow GProg to efficiently be able to compute the root keys and to replenish new circuits in nodes without having to remember all the existing labels. Since Sim can run in time propotional to the size of the database, it can simply remember all these values internally, and therefore the first step is to replace F with a truly random table that Sim keeps track of.

Next, we must simulate the garbled circuits one at a time. In order to do so, we order the circuits by the order in which they were evaluated, and then define a series of hybrids where hybrid i has the first i garbled circuits simulated. The hybrids are constructed so that the circuits are simulated in reverse order, such that the i-th circuit is simulated first, and so on, until the first circuit $\tilde{C}^0$ is simulated. At first glance, this appears to work, but there is actually a subtle issue. Each qKey of a circuit in a node circuit resides in several locations: the predecessor circuit in the same node, and inside several $\overline{\text{tKey}}$, oldLKey, oldRKey of circuits in its parent node. Indeed, if a full qKey appears anywhere (whether as a passed or hardwired value) in the current hybrid, then this causes a "circularity" issue. In order to overcome this barrier, we *pass* $\overline{\text{tKey}}$ from circuit to circuit instead of hardwiring them as seen in Figures 9 and 10.

This way, we can control which keys are present in a circuit. In particular, we drop consumed qKeys so that no future unevaluated circuit has them. However, we still need to hardwire new qKeys, but we only do so as needed as part of two new hardwired variables newLtKey and newRtKey. Then, whenever a circuit outputs some qKey for a child circuit, it also drops all qKeys inside $\overline{\text{tKey}}$ that are older than or equal to qKey. Note that we still maintain the exact same $\overline{\text{tKey}}$ size of 2κ keys, so this passing method will never accumulate too many keys. We formalize this construction by modifying how GData and C^{node} work in Figures 9 and 10, and the remainder of the construction is unchanged. It is straightforward to see that this neither affects correctness (only keys corresponding to consumed or soon-to-be-burned circuits will be dropped) nor asymptotic cost. We argue correctness as follows: within a node, the sequence of oldLKey and oldRKey within the circuits is an increasing sequence, and oldLKey strictly increases if we are going left, and oldRKey strictly increases if we are going right. Note that the $\overline{\text{tKey}}$ shifting corresponds precisely to the previous windows with the only difference being now there could be some keys set to $\perp$. Thus, the only way this scheme could be incorrect is if we attempt to assign a $\perp$ to oldLKey or oldRKey. But all the $\perp$ values correspond to indexes that are strictly less than the current oldLKey or oldRKey index, and therefore cannot be the new index.

We demonstrate a series of lemmas that ensures that when some circuit needs to be simulated, all appearances of its keys will have been in already been simulated or dropped. This strategy then allows the full simulation proof goes through as we will see in Section VII.

VII. Security Proof

In this section we prove the UMA2-security of the black-box garbled RAM (GData, GProg, GInput, GEval).

Theorem VII.1 (UMA2-security)**.** *Let F be a PRF and* (GCircuit, Eval, CircSim) *be a circuit garbling scheme, both of which can be built from any one-way function in black-box manner. Then our construction is a UMA2-secure garbled RAM scheme for uniform access programs running in total time $T < M$ making only black-box access to the underlying OWF.*

Proof. We first prove a lemma (Lemma VII.3) before proving our main theorem. For the lemma, we consider ourselves during the course of GEval, where we are about to evaluate some non-root node $\tilde{C}^{i,j,k}$. Our eventual goal is to show that all instances of $\text{qKey}^{i,j,k}$ are in previously evaluated (hence, will be simulated) circuits, and is not being passed as part of any $\overline{\text{tKey}}$.

Fact VII.2. *The rKey_{rec} to be consumed by $\tilde{C}^{i,j,k}$ was output by $\tilde{C}^{i,j,k-1}$, or initially stored in Tab(i,j) in the case where $k = 0$. The qKey_{q} used to evaluate $\tilde{C}^{i,j,k}$ was either output by (Case 1) $\tilde{C}^{i,j,k-1}$ or (Case 2) $\tilde{C}^{i-1,\lfloor j/2 \rfloor,k'}$ for some k'.*

To further pinpoint where qKeys are stored, we group the circuits in the parent node into three groups. We let k'_{min} be the smallest value such that $\lfloor (\frac{1}{2}+\epsilon)k'_{min} \rfloor + \kappa - 1 = k$, and k'_{max} be the largest value such that $\lfloor (\frac{1}{2}+\epsilon)k'_{max} \rfloor = k$. For a parent circuit $\tilde{C}^{i-1,\lfloor j/2 \rfloor,k'}$, we call it a *past* circuit if $k' < k'_{min}$, a *future* circuit if $k' > k'_{max}$, and a *present* circuit if $k'_{min} \le k' \le k'_{max}$.

We now state our main lemma.

Lemma VII.3. *Suppose during the execution of* GEval*, we are about to evaluate garbled circuit $\tilde{C}^{i,j,k}$. Let* qKey *denote $\text{qKey}^{i,j,k}$. Then all instances of* qKey *exist only in previously evaluated circuits.*

$C^{\mathsf{node}}[i, k, \mathsf{newLtKey}, \mathsf{newRtKey}, \mathsf{rKey}, \mathsf{qKey}]$
System parameters: ϵ (Will be set to $\frac{1}{\log M}$ as we will see later.)
Hardcoded parameters: $[i, k, \mathsf{newLtKey}, \mathsf{newRtKey}, \mathsf{rKey}, \mathsf{qKey}]$
Input: $(\mathsf{rec} = (\mathsf{lidx}, \mathsf{ridx}, \mathsf{oldLKey}, \mathsf{oldRKey}, \overline{\mathsf{tKey}}), \mathsf{q} = (goto, \mathsf{R/W}, L, z, \mathsf{cpuDKey}))$.

Set $\mathsf{p} := goto$ and $\mathsf{p}' := \left\lfloor \left(\frac{1}{2} + \epsilon\right) k \right\rfloor$.
Set $\mathsf{lidx}' := \mathsf{lidx}$ and $\mathsf{ridx}' := \mathsf{ridx}$. Set $\mathsf{oldLKey}' := \mathsf{oldLKey}$ and $\mathsf{oldRKey}' := \mathsf{oldRKey}$.
Define $ins(\overline{\mathsf{tKey}}, \mathsf{newLtKey}, \mathsf{newRtKey})$ to be the function that outputs $\overline{\mathsf{tKey}}$ with a possible shift: if $\left\lfloor \left(\frac{1}{2} + \epsilon\right)(k+1) \right\rfloor > \left\lfloor \left(\frac{1}{2} + \epsilon\right) k \right\rfloor$, shift $\overline{\mathsf{tKey}}$ to the left by 1 and set $\mathsf{tKey}[\kappa - 1] = \mathsf{newLtKey}, \mathsf{tKey}[2\kappa - 1] = \mathsf{newRtKey}$.
We now have three cases:

1) If $k < \mathsf{p} - 1$ then we output $(\mathsf{outrKey}, \mathsf{outqKey}) := (\mathsf{rKey}_{\mathsf{rec}'}, \mathsf{qKey}_{\mathsf{q}})$, where $\mathsf{rec}' := (\mathsf{lidx}', \mathsf{ridx}', \mathsf{oldLKey}', \mathsf{oldRKey}', \overline{\mathsf{tKey}'})$ where $\overline{\mathsf{tKey}'} = ins(\overline{\mathsf{tKey}}, \mathsf{newLtKey}, \mathsf{newRtKey})$.
2) If $k \geq \mathsf{p} + \kappa$ then abort with output `OVERCONSUMPTION-ERROR-I`.
3) If $\mathsf{p} - 1 \leq k < \mathsf{p} + \kappa$ then:
   a) If $L_i = 0$ then,
      i) If $\mathsf{lidx} < \mathsf{p}'$ then set $\mathsf{lidx}' := \mathsf{p}', goto' := \mathsf{p}'$ and $\mathsf{oldLKey}' := \mathsf{tKey}[0]$. Else set $\mathsf{lidx}' := \mathsf{lidx} + 1, goto' := \mathsf{lidx}'$ and if $\mathsf{lidx}' < \mathsf{p}' + \kappa$ then set $\mathsf{oldLKey}' := \mathsf{tKey}[\mathsf{lidx}' - \mathsf{p}']$ else abort with `OVERCONSUMPTION-ERROR-II`.
      ii) Set $\mathsf{tKey}[v] := \perp$ for all $v < \mathsf{lidx}' - p'$. Set $\overline{\mathsf{tKey}'} = ins(\overline{\mathsf{tKey}}, \mathsf{newLtKey}, \mathsf{newRtKey})$.
      iii) Set $\mathsf{outqKey} := \mathsf{oldLKey}_{\mathsf{q}'}$, where $\mathsf{q}' := \mathsf{q}$ but with $goto'$ replacing $goto$.
      else
      i) If $\mathsf{ridx} < \mathsf{p}'$ then set $\mathsf{ridx}' := \mathsf{p}', goto' := \mathsf{p}'$ and $\mathsf{oldRKey}' := \mathsf{tKey}[\kappa]$. Else set $\mathsf{ridx}' := \mathsf{ridx} + 1, goto' := \mathsf{ridx}'$ and if $\mathsf{ridx}' < \mathsf{p}' + \kappa$ then set $\mathsf{oldRKey}' := \mathsf{tKey}[\kappa + \mathsf{ridx}' - \mathsf{p}']$ else abort with `OVERCONSUMPTION-ERROR-II`.
      ii) Set $\mathsf{tKey}[\kappa + v] := \perp$ for all $v < \mathsf{ridx}' - p'$. Set $\overline{\mathsf{tKey}'} = ins(\overline{\mathsf{tKey}}, \mathsf{newLtKey}, \mathsf{newRtKey})$.
      iii) Set $\mathsf{outqKey} := \mathsf{oldRKey}_{\mathsf{q}'}$, where $\mathsf{q}' := \mathsf{q}$ but with $goto'$ replacing $goto$.
   b) Set $\mathsf{outrKey} := \mathsf{rKey}_{\mathsf{rec}'}$ where $\mathsf{rec}' := (\mathsf{lidx}', \mathsf{ridx}', \mathsf{oldLKey}', \mathsf{oldRKey}', \overline{\mathsf{tKey}'})$ and output $(\mathsf{outrKey}, \mathsf{outqKey})$.

Figure 9. Formal description of the nonleaf Memory Circuit with key passing.

Proof: We let $k^\star$ denote the index of the last parent circuit that evaluated prior to our current circuit, i.e. $\tilde{C}^\star = \tilde{C}^{i-1, \lfloor j/2 \rfloor, k^\star}$ was the last circuit to be evalauted at level $i - 1$. WLOG assume that the current circuit is the left child of the parent. Observe that qKey only occurs in the following locations: the predecessor circuit, inside $\mathsf{newLtKey}$ of the final "past" parent, or inside some "current" or "future" parent's $\mathsf{oldLKey}$ or $\overline{\mathsf{tKey}}$. Since the predecessor circuit must be evaluated already, we only need to check the existence of qKey inside one or more of my parent circuits.

Let lidx be the left index (implicitly) passed into $\tilde{C}^\star$, and let lidx' be the left index (implicitly) output by it.

Observe that by definition, qKey is not in the $\overline{\mathsf{tKey}}$ of any past or future parent. In particular, it can only be included inside $\overline{\mathsf{tKey}}$ when being inserted as a $\mathsf{newLtKey}$, and once it is removed it can never be present again in any future parent's $\overline{\mathsf{tKey}}$. Note that qKey may still be inside $\mathsf{newLtKey}$ of a past parent or $\mathsf{oldLKey}$ of a future parent. Furthermore, all parent circuits with index $k' \leq k^\star$ have been evaluated, and thus we only need to argue that no (unevaluated) parent circuit $k' > k^\star$ contains qKey as either $\overline{\mathsf{tKey}}$, $\mathsf{newLtKey}$, or $\mathsf{oldLKey}$.

We analyse the following six cases:

[Case 1A] The predecessor circuit $\tilde{C}^{i,j,k-1}$ output my $\mathsf{qKey}_{\mathsf{q}}$, and $k^\star$ belongs to a past parent.
[Case 1B] The predecessor circuit $\tilde{C}^{i,j,k-1}$ output my $\mathsf{qKey}_{\mathsf{q}}$, and $k^\star$ belongs to a present parent.
[Case 1C] The predecessor circuit $\tilde{C}^{i,j,k-1}$ output my $\mathsf{qKey}_{\mathsf{q}}$, and $k^\star$ belongs to a future parent.
[Case 2A] The parent circuit $\tilde{C}^{i-1, \lfloor j/2 \rfloor, k^\star}$ output my $\mathsf{qKey}_{\mathsf{q}}$, and $k^\star$ belongs to a past parent.
[Case 2B] The parent circuit $\tilde{C}^{i-1, \lfloor j/2 \rfloor, k^\star}$ output my $\mathsf{qKey}_{\mathsf{q}}$, and $k^\star$ belongs to a present parent.
[Case 2C] The parent circuit $\tilde{C}^{i-1, \lfloor j/2 \rfloor, k^\star}$ output my $\mathsf{qKey}_{\mathsf{q}}$, and $k^\star$ belongs to a future parent.

CASE 1A. This case cannot occur. Since $\tilde{C}^\star$ was a past parent, by definition we must have $\lfloor (\frac{1}{2} + \epsilon) k^\star \rfloor + \kappa - 1 < k$. Since qKey was passed from the predecessor circuit, it must have taken the branch where $k - 1 < goto' - 1 = \lfloor (\frac{1}{2} + \epsilon) k^\star \rfloor - 1$. Combining these two inequalities yields $\kappa < 2$ which is a contradiction.

CASE 1B,1C. Note that $k - 1 < goto' - 1$ still holds. We know that $\mathsf{lidx}' \geq goto'$, so we have that $\mathsf{lidx}' > k$. Since $\mathsf{oldLKey}'$ is the key for circuit lidx', it cannot be the qKey for k. Furthermore, all keys inside $\overline{\mathsf{tKey}}$ with index less than lidx' have

The algorithm $\mathsf{GData}(1^\kappa, D)$ proceeds as follows. Without loss of generality we assume that $M = 2^d$ (where $M = |D|$) where d is a positive integer. We calculate $\epsilon = \frac{1}{\log M}$.We set $K_0 = M$, and for each $0 < i \in [d+1]$ and set $K_i = \left\lfloor \left(\frac{1}{2} + \epsilon\right) K_{i-1} \right\rfloor + \kappa$.

1) Let $s \leftarrow \{0,1\}^\kappa$.
2) Any $\mathsf{dKey}^{d,j,k}$ needed in the computation below is obtained as $F_s(\mathtt{data}||d||j||k)$. Similarly for any i, j, k, $\mathsf{rKey}^{i,j,k} := F_s(\mathtt{rec}||i||j||k)$ and $\mathsf{qKey}^{i,j,k} := F_s(\mathtt{query}||i||j||k)$.
   Set
$$\overline{\mathsf{tKey}}^{i,j,0} := \left\{ \underbrace{\left\{\mathsf{qKey}^{i+1,2j,l}\right\}_{l\in[\kappa]}}_{\text{left}}, \underbrace{\left\{\mathsf{qKey}^{i+1,2j+1,l}\right\}_{l\in[\kappa]}}_{\text{right}} \right\}.$$
   and if $\left\lfloor \left(\frac{1}{2} + \epsilon\right)(k+1) \right\rfloor > \left\lfloor \left(\frac{1}{2} + \epsilon\right)(k) \right\rfloor$, then set
$$\mathsf{newLtKey}^{i,j,k} = \mathsf{qKey}^{i+1,2j,\left\lfloor \left(\frac{1}{2}+\epsilon\right)(k+1)\right\rfloor + \kappa - 1}$$
$$\mathsf{newRtKey}^{i,j,k} = \mathsf{qKey}^{i+1,2j+1,\left\lfloor \left(\frac{1}{2}+\epsilon\right)(k+1)\right\rfloor + \kappa - 1}$$
   , otherwise set $\mathsf{newLtKey}^{i,j,k} = \mathsf{newRtKey}^{i,j,k} = \perp$.
3) For all $j \in [2^d], k \in [K_d]$,
   $\tilde{C}^{d,j,k} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{leaf}}\left[d, k, \mathsf{dKey}^{d,j,k+1}, \mathsf{qKey}^{d,j,k+1}\right], \mathsf{dKey}^{d,j,k}, \mathsf{qKey}^{d,j,k}\right)$.
4) For all $i \in [d], j \in [2^i], k \in [K_i]$,
   $\tilde{C}^{i,j,k} \leftarrow \mathsf{GCircuit}(1^\kappa, \mathsf{C}^{\mathsf{node}}\left[i, k, \mathsf{newLtKey}^{i,j,k}, \mathsf{newRtKey}^{i,j,k}, \mathsf{rKey}^{i,j,k+1}, \mathsf{qKey}^{i,j,k+1}\right], \mathsf{rKey}^{i,j,k}, \mathsf{qKey}^{i,j,k})$.
5) For all $j \in [2^d]$, set $\mathsf{Tab}(d, j) = \mathsf{dKey}^{d,j,0}_{D[j]}$.
6) For all $i \in [d], j \in [2^i]$, set $\mathsf{Tab}(i, j) := \mathsf{rKey}^{i,j,0}_{\mathsf{rec}^{i,j,0}}$, where $\mathsf{rec}^{i,j,0} := (0, 0, \mathsf{qKey}^{i+1,2j,0}, \mathsf{qKey}^{i+1,2j+1,0}, \overline{\mathsf{tKey}}^{i,j,0})$.
7) Output $\tilde{D} := \left(\left\{\tilde{C}^{i,j,k}\right\}_{i\in[d+1], j\in[2^i], k\in[K_i]}, \{\mathsf{Tab}(i,j)\}_{i\in[d+1], j\in[2^i]}\right)$ and s.

Figure 10. Formal description of GData with passed keys.

been set to $\perp$ by $\tilde{C}^\star$ so no unevaluated parent circuit can have the current qKey as part of $\overline{\mathsf{tKey}}$ or oldLKey. Finally, qKey appearing as newLtKey can only occur in a past parent, which has already been evaluated in this case.

CASE 2A. This case cannot occur. By the definition of C^{node}, the only way the parent circuit could output my qKey_q directly is if it is held as oldLKey. However, the oldLKey is only assigned due to the value contained in an older tKey in some parent circuit $k' \le k^\star$. The indices k of any parent $k' \le k^\star$ parents is at most $\lfloor(\frac{1}{2}+\epsilon)k'\rfloor + \kappa - 1 \le \lfloor(\frac{1}{2}+\epsilon)k^\star\rfloor + \kappa - 1 < k$ by definition of $k^\star$ being a past parent. Therefore, qKey could not have been output by any past parent circuit.

CASE 2B. In this case, $k^\star$ belongs to a present circuit that was evaluated. Note that $\tilde{C}^\star$ replaced its oldLKey = qKey with some new oldLKey$'$ which corresponds to the lidx$'$-th circuit at level i. Since $k = \mathsf{lidx} < \mathsf{lidx}'$ and all tKey$[v]$ is set to $\perp$ for $v < \mathsf{lidx}' - \mathsf{p}'$, the current qKey was removed from $\overline{\mathsf{tKey}}$ by $\tilde{C}^\star$ and hence all successor parent circuits' $\overline{\mathsf{tKey}}$ do not contain qKey. Furthermore, oldLKey can only be updated by $\overline{\mathsf{tKey}}$ and $\tilde{C}^\star$ does not set the updated oldLKey$'$ to qKey, and no parent circuit $k' > k^\star$ can set oldLKey to qKey since it is no longer contained in any of their $\overline{\mathsf{tKey}}$ values. Finally, qKey appearing as newLtKey can only occur in a past parent, which has already been evaluated in this case.

CASE 2C. Because $k^\star$ belongs to a future parent that was evaluated, it must be the case that all past and present parents have already been evaluated. We check that qKey does not exist in any unevaluated parent circuit's $\overline{\mathsf{tKey}}$ or oldLKey: all parent circuits $k' \le k^\star$ have been evaluated, $\tilde{C}^\star$ was evaluated and it output and replaced the qKey sitting in oldLKey with some tKey. Since $\tilde{C}^\star$ and all its successors are future parents, none of them have qKey inside its $\overline{\mathsf{tKey}}$ and thus oldLKey would never contain qKey. Finally, qKey appearing as newLtKey can only occur in a past parent, which has already been evaluated in this case.

■

We now proceed to prove the theorem. Let CircSim be the garbled circuit simulator. Suppose in the real execution, a total of w circuits are evaluated by GEval. We construct Sim and then give a series of hybrids $\hat{H}^0, H^0, \ldots, H^w, \hat{H}^w$ such that the first hybrid outputs the $(\tilde{D}, \tilde{P}, \tilde{x})$ of the is the real execution and the last hybrid is the output of Sim, which we

will define. H^0 is the real execution with the PRF F replaced with a uniform random function (where previously evaluated values are tabulated). Since the PRF key is not used in evaluation, we immediately obtain $\hat{H}^0 \overset{\text{comp}}{\approx} H^0$.

Our goal is to build a garbled memory, program, and input that is indistinguishable from the real one. Since we know exactly the size and running time and memory access, we can allocate the exact correct amount of placeholder garbled circuits, initially set to $\perp$. The simulator considers the sequence of circuits (starting from 1) that would have been evaluated given $\mathsf{MemAccess}$. This sequence is entirely deterministic and therefore we let $S_1, \ldots, S_w$ be this sequence of circuits, e.g. $S_1 = \tilde{C}^0$(the first CPU circuit), $S_2 = \tilde{C}^{0,0,0}$(the first root circuit), ... The idea is to have H^u simulate the first u of these circuits, and generate all other circuits as in the real execution.

Hybrid Definition: $(\tilde{D}, \tilde{P}, \tilde{x}) \leftarrow H^u$

The hybrid H^u proceeds as follows: For each circuit not in $S_1, \ldots, S_u$, generate it as you would in the real execution, and for each circuit $S_u, \ldots, S_1$ (in that order) we simulate the circuit using $\mathsf{CircSim}$ by giving it as output what it would have generated in the real execution or what was provided as the simulated input labels. Note that this may use information about the database D and the input x, and our goal is to show that at the very end, Sim will not need this information.

We now show $H^{u-1} \overset{\text{comp}}{\approx} H^u$. There are several cases: when S_u is a non-root node, when S_u is a root node, and when S_u is a CPU step circuit. In the first case, we must argue that one can replace S_u from a real distribution to one output by $\mathsf{CircSim}$. In order to do so, we must show that its input keys are independent of the rest of the output. Its garbled inputs are $\mathsf{rKey}_{\mathsf{rec}}$ and $\mathsf{qKey}_{\mathsf{q}}$. rKey only existed in its predecessor circuit, which was simulated since it was executed in some hybrid $u' < u$. Furthermore, by Lemma VII.3, all instances of qKey only exist in previously evaluated circuits, hence they were also simulated out in some earlier hybrid. Therefore, any distinguisher of H^{u-1} and H^u can be used to distinguish between the output of $\mathsf{CircSim}$ and a real garbling.

When S_u is a root node, the only circuit that has its rKey was its predecessor, and the only circuits that have its qKey are its predecessor or the CPU step circuit that invoked it. Both of these circuits were simulated in a earlier hybrid, and so once again any distinguisher of H^{u-1} and H^u can be used to distinguish between the output of $\mathsf{CircSim}$ and a real garbling.

Finally, if S_u is a CPU step circuit, the only circuit that has its $\mathsf{cpuSKey}$ was its predecessor (or the initial garbled input), but its $\mathsf{cpuDKey}$ was passed around across the entire tree starting from its predecessor. However, again, these were all simulated in an earlier hybrid, so again, any distinguisher of H^{u-1} and H^u can be used to distinguish between the output of $\mathsf{CircSim}$ and a real garbling.

Finally, we mention how to handle unevaluated circuits in hybrid $\hat{H}^w$. Note that the security definition of $\mathsf{CircSim}$ does not deal with partial inputs, though this can be handled generically as follows. We encrypt each circuit using a semantically secure symmetric key-encryption scheme with a fresh key for each circuit and secret share the key into two portions. We augment each $\mathsf{rKey}/\mathsf{dKey}$ by giving it one share (in the clear), and the qKey will have the other share. In unevaluated circuits, the qKey never appears, so the secret encryption key is information theoretically hidden, and thus by the semantic security of the encryption scheme, we can replace all unevaluated circuits with encryptions of zero. This is formally stated and proven in the full version [GLO15].

Then our simulator $\mathsf{Sim}(1^\kappa, 1^M, 1^t, y, 1^D, \mathsf{MemAccess} = \{L^\tau, z^{\mathsf{read},\tau}, z^{\mathsf{write},\tau}\}_{\tau=0,\ldots,t-1})$ can output the distribution $\hat{H}^w$ without access to D or x. We see this as follows: the simulator, given $\mathsf{MemAccess}$ can determine the sequence $S_1, \ldots, S_w$. The simulator starts by simulating all unevaluted circuits by replacing them with encryptions of zero. It then simulates the S_u in reverse order, starting with simulating S_w using the output y, and then working backwards simulates further ones ensuring that their output is set to the appropriate inputs.

□

VIII. Full Security and Replenishing

In this section, we prove the full security by showing how to compile any UMA2-secure GRAM scheme with statistical ORAM. First, we show how to extend our construction to running times beyond M by replenishing.

A. Circuit Replenishing

Although we observed that "dynamic" circuit replenishing is potentially problematic, here we give a method of allowing GProg to replenish circuits. Note that in GData there was no need to generate some fixed amount of circuits at the root, but it was bounded to be proportional to M in order for it to not run too long. However, using the exact same template, an exponential amount of circuits could be generated: as long as the domain of the PRF is not exhausted, one can always generate more circuit labels that follow this exact pattern.

Using this observation, we can then view our circuit replenishing as a way to amortize this process rather than making dynamic replacements on the fly. That is to say, when we make new circuits, they will be concatenated on to the end of the sequence of circuits of each node. In order to replenish, we give a replenishment strategy so that GProg will be augmented

to perform the following functionality. A program at time m running for t steps will replenish some $t \cdot \mathsf{poly}(\log M, \log T, \kappa)$ number of circuits in such a way that after M steps, a total of $S = \sum_{i=0}^{d}(2^i K_i)$ circuits will be replenished, where K_i is as defined in GData, thus providing us with a number of new circuits that is as large as the original number of original circuits provided. We provide the details on how this is done in the full version [GLO15].

B. Compiling with Statistical ORAM to get full security

In order to achieve this, we show how to compile our UMA2-secure GRAM scheme with a statistical ORAM that has uniform access pattern to achieve a secure GRAM scheme. However, schemes such as [SvDS+13] are tree-based and have uniform access pattern only on each level of their tree. That is to say, on each level, the access pattern is uniform, though not necessarily on the entire tree.

Leveled Memory: In order to combat this issue, we make several *independent* copies of memory, each corresponding to a level in the ORAM tree. Then each CPU step will have the key corresponding to the memory block of one level of the ORAM tree. Indeed, this is a generic method of handling leveled memory, though one possible avenue of concrete benefits is "marrying" together the underlying ORAM tree with our GRAM tree. We give a more formal description in the full version [GLO15].

The last step is to combine ORAM with UMA2-secure GRAM to obtain fully secure GRAM. The proof is nearly identical to extending UMA-secure GRAM to fully secure GRAM, so we paraphrase previous works [LO13b], [GHL+14], [GLOS15] and we show this in the full version [GLO15].

Putting it all together, we obtain our main theorem.

Theorem VIII.1 (Full security)**.** *Assuming only the existence of one-way functions, there exists a secure black-box garbled RAM scheme for arbitrary RAM programs. The size of the garbled database is $\tilde{O}(|D|)$, size of the garbled input is $\tilde{O}(|x|)$ and the size of the garbled program and its evaluation time is $\tilde{O}(T)$ where T is the running time of program P. Here $\tilde{O}(\cdot)$ ignores $\mathsf{poly}(\log T, \log |D|, \kappa)$ factors where κ is the security parameter. Furthermore, because garbled RAM trivially implies one-way functions, there is a black-box equivalence of the existence of one-way functions and garbled RAM.*

Additional observations: Instead of storing data only at the leaves, we can store the data at all levels of the tree and pull an entire path of values from the tree down into the CPU step. This is conducive to certain ORAM schemes (e.g. Path ORAM [SvDS+13]) which also follow this nature and can be used to obtain additional savings. Furthermore, in our construction, one can consider handling a non-uniform distribution of memory accesses. As long as the distribution of leaf accesses are data-independent and known in advance, we can assign to each leaf a probability it is accessed. Then each parent inherits a probability that is the sum of its two childrens' probabilities. Based on this new distribution, one can provide a different number of circuits per node as according to this distribution. This would lead to a concrete efficiency improvement for GRAM in the case of certain oblvious algorithms with simple CPU steps and access patterns that are distributed in some known fashion.

ACKNOWLEDGMENTS

We thank Alessandra Scafuro for her contribution during the initial stages of the project. First author is supported by NSF CRII Award 1464397. Second and third authors are 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

[ACG+14] Prabhanjan Ananth, Nishanth Chandran, Vipul Goyal, Bhavana Kanukurthi, and Rafail Ostrovsky. Achieving privacy in verifiable computation with multiple servers - without FHE and without pre-processing. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 149–166. Springer, March 2014.

[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.

[Ajt10] Miklós Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In Leonard J. Schulman, editor, *42nd ACM STOC*, pages 181–190. ACM Press, June 2010.

[Bea96] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *28th ACM STOC*, pages 479–488. ACM Press, May 1996.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *STOC*, pages 103–112, 1988.

[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/2014/771.

[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/2014/769.

[CLP14] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 62–81. Springer, December 2014.

[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*, volume 6597 of *LNCS*, pages 144–163. Springer, March 2011.

[FLS99] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple non-interactive zero knowledge proofs under general assumptions. *SIAM Journal of Computing*, 29(1):1–28, 1999.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 1–17. Springer, May 2013.

[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 FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

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

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *55th FOCS*, pages 404–413. IEEE Computer Society Press, October 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, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 536–553. Springer, August 2013.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. Cryptology ePrint Archive, Report 2015/307, 2015. http://eprint.iacr.org/2015/307.

[GLOS14] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. Cryptology ePrint Archive, Report 2014/941, 2014. http://eprint.iacr.org/2014/941.

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

[GLOV12] Vipul Goyal, Chen-Kuei Lee, Rafail Ostrovsky, and Ivan Visconti. Constructing non-malleable commitments: A black-box approach. In *53rd FOCS*, pages 51–60. IEEE Computer Society Press, October 2012.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[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 ACM STOC*, pages 182–194. ACM Press, May 1987.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for np. In *Proceedings of Eurocrypt 2006, volume 4004 of LNCS*, pages 339–358. Springer, 2006.

[GOSV14] Vipul Goyal, Rafail Ostrovsky, Alessandra Scafuro, and Ivan Visconti. Black-box non-black-box zero knowledge. In David B. Shmoys, editor, *46th ACM STOC*, pages 515–524. ACM Press, May / June 2014.

[IKLP06] Yuval Ishai, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. Black-box constructions for secure computation. In Jon M. Kleinberg, editor, *38th ACM STOC*, pages 99–108. ACM Press, May 2006.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, August 2003.

[IR89] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *21st ACM STOC*, pages 44–61. ACM Press, May 1989.

[IR90] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In Shafi Goldwasser, editor, *CRYPTO'88*, volume 403 of *LNCS*, pages 8–26. Springer, August 1990.

[LO13a] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 377–396. Springer, March 2013.

[LO13b] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 719–734. Springer, May 2013.

[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/2014/766.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th ACM STOC*, pages 294–303. ACM Press, May 1997.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd ACM STOC*, pages 514–523. ACM Press, May 1990.

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

[PW09] Rafael Pass and Hoeteck Wee. Black-box constructions of two-party protocols from one-way functions. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 403–418. Springer, March 2009.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, December 2011.

[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*, pages 299–310. ACM Press, November 2013.

[Wee10] Hoeteck Wee. Black-box, round-efficient secure computation via non-malleability amplification. In *51st FOCS*, pages 531–540. IEEE Computer Society Press, October 2010.

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 191–202. ACM Press, November 2014.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.