Garbled RAM Revisited

Craig Gentry[1], Shai Halevi[1], Steve Lu[2], Rafail Ostrovsky[2], Mariana Raykova[3,⋆], and Daniel Wichs[4,⋆⋆]

[1] IBM Research
[2] UCLA
[3] SRI
[4] Northeastern University

Abstract. The notion of *garbled random-access machines* (garbled RAMs) was introduced by Lu and Ostrovsky (Eurocrypt 2013). It can be seen as an analogue of Yao's garbled circuits, that allows a user to garble a RAM program directly, without performing the expensive step of converting it into a circuit. In particular, the size of the garbled program and the time it takes to create and evaluate it are only proportional to its running time on a RAM rather than its circuit size. Lu and Ostrovsky gave a candidate construction of this primitive based on pseudo-random functions (PRFs).

The starting point of this work is pointing out a subtle circularity hardness assumption in the Lu-Ostrovsky construction. Specifically, the construction requires a complex "circular" security assumption on the underlying Yao garbled circuits and PRFs. We then proceed to abstract, simplify and generalize the main ideas behind the Lu-Ostrovsky construction, and show two alternatives constructions that overcome the circularity of assumptions. Our first construction breaks the circularity by replacing the PRF-based encryption in the Lu-Ostrovsky construction by *identity-based encryption (IBE)*. The result retains the same asymptotic performance characteristics of the original Lu-Ostrovsky construction, namely overhead of $O(\mathsf{poly}(\kappa)\mathsf{polylog}(n))$ (with κ the security parameter and n the data size). Our second construction breaks the circularity assuming only the existence of one way functions, but with overhead $O(\mathsf{poly}(\kappa)n^{\varepsilon})$ for any constant $\varepsilon > 0$. This construction works by adaptively "revoking" the PRFs at selected points, and using a delicate recursion argument to get successively better performance characteristics. It remains as an interesting open problem to achieve an overhead of $\mathsf{poly}(\kappa)\mathsf{polylog}(n)$ assuming only the existence of one-way functions.

1 Introduction

Garbled Circuits. Since their introduction by Yao [19], garbled circuits have found countless applications in cryptography, most notably for secure computation. On a basic level, garbled circuits allow a user to convert a circuit C into

⋆ Research conducted in part while at IBM Research.
⋆⋆ Research conducted in part while visiting IBM Research.

P.Q. Nguyen and E. Oswald (Eds.): EUROCRYPT 2014, LNCS 8441, pp. 405–422, 2014.

a garbled version $\tilde{C}$ and an input x into a garbled version $\tilde{x}$, so that $\tilde{C}$ can be evaluated on $\tilde{x}$ to reveal the output $C(x)$, but nothing else is revealed. As with most secure computation protocols, this technique crucially works at the level of "circuits" and the first step toward using it is to convert a desired program into a circuit representation.

Circuits vs. RAMs. Converting a program into a circuit often presents a major source of inefficiency. We naturally think of programs in the the *random-access machine* (RAM) model of computation. It is known that a RAM with run-time T can be converted into a Turing Machine with run-time $O(T^3)$ which can in turn be converted into the circuit of size $O(T^3 \log T)$ [8,16]. This is a significant amount of overhead. Perhaps an even more striking efficiency loss occurs in the setting of "big data", where the data is given in random-access memory. In this case, efficient programs can run in time which is sub-linear in the size of the data (e.g., binary search), but converting any such a program into a circuit representation incurs a cost which is (at the very least) linear in the size of the data. This exponential gap can mean the difference between an efficient Internet search and having to read the entire Internet!

Garbled RAMs. Motivated by the above considerations, Lu and Ostrovsky [14] proposed the notion of a *garbled RAM*, whose goal is to garble a RAM program directly without first converting it into a circuit. In particular, the size of the garbled program as well as the evaluation time should only be proportional to the running-time of the program on a RAM (up to poly-logarithmic factors), rather than the size of its circuit representation.

In more detail, we will use the notation $P^D(x)$ to denote the execution of some program P with random-access memory initially containing some data D and a "short" input x (e.g., P could be some complex query over a database D with search-terms x). A garbled RAM scheme can be used to garble the data D into $\tilde{D}$, the program P into $\tilde{P}$, and the input x into $\tilde{x}$ in such a way that $\tilde{P}, \tilde{D}, \tilde{x}$ reveals $P^D(x)$, but nothing else is revealed. Furthermore, the size of the garbled data $\tilde{D}$ is only proportional to that of D, the size of $\tilde{x}$ is only proportional to that of x, and the size and evaluation-time of the garbled program $\tilde{P}$ are only proportional to the run-time of $P^D(x)$ on a RAM.

Lu and Ostrovsky proposed a construction of garbled RAMs, relying on a clever use of Yao garbled circuits and oblivious RAM (ORAM), and using for security only pseudo-random functions (PRFs) (which can be constructed from any one-way function).

A Circularity Problem. It turns out that the Lu-Ostrovsky construction has a subtle yet difficult-to-overcome issue that prevents a proof of security from going through, in that it requires a complex "circular" use of Yao garbled circuits and PRF-based encryption. To understand the issue, recall that Yao garbled circuits assign two labels for each wire, corresponding to bits $0, 1$, and security relies heavily on the evaluator only learning one of these two labels. The Lu-Ostrovsky construction provides encryptions of *both* labels of an input wire w,

under some secret-key K, and this secret key K is also hard-coded into the description of the circuit itself. This introduces the following circularity: to use the security of the encryption scheme we must rely on the security of the garbled circuit to hide the key K, but to use the security of the garbled circuit we must rely on the security of the encryption scheme so that the attacker cannot learn *both* wire labels. We emphasize that we do not have a concrete attack on the construction of Lu and Ostrovsky, and it may even seem reasonable to conjecture its security when instantiated with real-world primitives (e.g., AES). Unfortunately, we don't see much hope for proving the security of the scheme under standard assumptions. One could draw an analogy to other "subtle" difficulties in cryptography such as circular security [5,17], selective-opening security [4,2], or adaptively-chosen inputs of garbled circuits [3], where it may be reasonable to assume that standard constructions are secure (and it's a challenge to come up with insecure counterexamples), but it doesn't seem that one can prove security of standard constructions under standard assumptions.

Our Results. In this work we abstracts, simplifies, and generalizes the main ideas behind the Lu-Ostrovsky construction, and give two solutions to the circularity problem. Our first construction essentially replaces the PRF-based encryption in the construction from [14] by *identity-based encryption* (IBE). This breaks the circularity since we only need to embed in the circuit the public key of the IBE, not the secret key. This scheme can be proved secure under the security of the underlying IBE (and garbled circuits), and its overhead is only $\mathsf{poly}(\kappa)\mathsf{polylog}(n)$, where κ is the security parameter and n is the size of the data. (The overhead is measured as the evaluation time of a garbled programs vs. the original program.) This construction is described in detail in [9].

In the second construction, we break the circularity using *revocable PRFs* that enables adaptive revocation of the ability to compute the PRF on certain values.[1] Namely, from a PRF key K and a subset X of the domain, we can construct a weaker key K_X that enables the computation of $F_K(\cdot)$ on all the domain *except for* X, and the values $F_K(x)$ for $x \in X$ are pseudo-random even given K_X. Importantly for our application, we also need successive revocation, i.e. from K_X and some X' we should be able to to generate $K_{X \cup X'}$. Such revocable PRFs can be constructed based on the Goldreich-Goldwasser-Micali [10] PRF, where the size of the key K_X is at most $\kappa \cdot |X| \log N$ (with κ the security parameter and N the domain size).

We use revocable PRFs to break the circularity as follows: whenever we use some $F_K(x)$ in the encryption of the label values on the input wire w, we make sure to embed in the circuit itself not the original key K but rather the weaker key K_X (with $x \in X$), so the encryption remain secure even if K_X is known. A naive use of this technique yields a trivial scheme with overhead $\mathsf{poly}(\kappa) \cdot n$, which is no better than using circuits. However we show how to periodically refresh the keys to reduce the overhead to roughly $\mathsf{poly}(\kappa)\sqrt{n}$, and then use a

[1] This notion is similar to punctured PRFs [18], delegatable PRFs [13], functional PRFs [7], and constrained PRFs [6], see more details in Definition 3.

recursive strategy to reduce it further to $\mathsf{poly}(\kappa) \cdot min(t, n^{\varepsilon})$ for any constant $\varepsilon > 0$ (where n is the data size and t is the running time). This construction is described in detail in [15].

Reusable/Persistent Data. We also carefully define and prove the security of an important use-case of garbled RAMs, where the garbled memory data can be reused across multiple program executions. If a program updates some location in memory, these changes will persist for future program executions and cannot be "rolled back" by the adversarial evaluator. For example, consider a client that garbles some huge database D and outsources the garbled version $\tilde{D}$ to a remote server. Later, the client can sequentially garble arbitrary database queries so as to allow the server to execute exactly the garbled query on the garbled database but not learn anything else. If the query updates some values in the database, these changes will persist for the future. The running time of the client and server per database query is only proportional to the RAM run-time of the query.[2] Prior to garbled RAMs, this could be done using oblivious RAM (ORAM) but would have required numerous rounds of interaction between the client and the server per database query. With garbled RAMs, the solution becomes non-interactive. This use-case was already envisioned by Lu and Ostrovsky [14], but we proceed to define and analyze it formally.

Worst-Case versus Per-Instance Running Time, Universal Programs, and Output Privacy. As was noted in the CRYPTO 2013 work of Goldwasser et al. [11], the power of secure computation on Turing Machines and RAM programs over that of circuits is that for algorithms with very different worst-case and average-case running times, the circuit must be of worst-case size. Randomized algorithms such as Las Vegas algorithms or even heuristically good-on-average programs would benefit greatly if the online running time of the secure computation ran in time proportional to that particular instance. In our solution, though we have an upper bound T on the number of execution steps of the algorithm which affects the offline time and space, the online evaluation can have a CPU step output "halt" in the clear when the program has halted and the evaluator will then only run in time depending on this particular input.

In order to further mask the program, one can consider a T time-bounded universal program u_T, which takes as input the code of a program π and an input for that program. One can also provide an auxiliary mask so that the output of P is blinded by this value (such a modification has appeared in the literature, see, e.g. [1]).

Organization. We describe our notations for RAM computation and define garbled RAM in Section 2. We then give a high-level description of the Lu-Ostrovsky

[2] In contrast to schemes for *outsourcing* computation, the client here does not save on work, but only saves on storage. In particular, only the garbled data $\tilde{D}$ is reusable, but the garbled program $\tilde{P}$ can still only be evaluated on a single garbled input $\tilde{x}$; the client must garble a fresh program for each execution, which requires time proportional to that of the execution.

construction in Section 3, along with an explanation of the "circularity" issue. In Section 4 we present our IBE-based solution, and in Section 5 we describe our solution based on one-way functions.

2 RAM Computation and Garbled RAM

Notation for RAM Computation. Consider a program P that has random-access to a memory of size n, which may initially contain some data $D \in \{0,1\}^n$, and a "short" input x. [3] We use the notation $P^D(x)$ to denote the execution of such program. The program can read/write to various locations in memory throughout the execution. 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 a useful example to keep in mind throughout this work, imagine that D is a huge database and the programs P_i are database queries that can read and possibly write to the database and are parameterized by some values x_i.

CPU-Step Circuit. A useful representation of a RAM program P is through a small *CPU-Step Circuit* which executes a single CPU step:

$$C^P_{\mathsf{CPU}}(\mathtt{state}, b^{\mathsf{read}}) = (\mathtt{state}', i^{\mathsf{read}}, i^{\mathsf{write}}, b^{\mathsf{write}})$$

This circuit takes as input the current CPU $\mathtt{state}$ and a bit b^{read} residing in the the last read memory location. It outputs an updated $\mathtt{state}'$, the next location to read $i^{\mathsf{read}} \in [n]$, a location to write to $i^{\mathsf{write}} \in [n] \cup \{\perp\}$ (where $\perp$ values are ignored), a bit b^{write} to write into that location.

The computation $P^D(x)$ starts in the initial state $\mathtt{state}_1 = x$, corresponding to the "short input" and by convention we will set the initial read bit to $b_1^{\mathsf{read}} := 0$. In each step j, the computation proceeds by running $C^P_{\mathsf{CPU}}(\mathtt{state}_j, b_j^{\mathsf{read}}) = (\mathtt{state}_{j+1}, i^{\mathsf{read}}, i^{\mathsf{write}}, b^{\mathsf{write}})$. We first read the requested location i^{read} by setting $b_{j+1}^{\mathsf{read}} := D[i^{\mathsf{read}}]$ and, if $i^{\mathsf{write}} \neq \perp$, we write to the location by setting $D[i^{\mathsf{write}}] := b^{\mathsf{write}}$. The value $y = \mathtt{state}$ output by the last CPU step serves as the output of the computation.

We say that a program P has ***read-only*** memory access, if it never overwrites any values in memory. In particular, using the above notation, the outputs of C^P_{CPU} always set $i^{\mathsf{write}} = \perp$.

2.1 Defining Garbled RAM

We consider a setting where the memory data D is garbled once, and then many different garbled programs can be executed sequentially with the memory

[3] 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.

changes persisting from one execution to the next. We stress that each garbled program $\tilde{P}_i$ can only be executed on a *single* garbled input $\tilde{x}_i$. In other words, although the garbled data is reusable and allows for the execution of many programs, the garbled programs are *not* reusable. The programs can only be executed in the specified order and are not "interchangeable". Therefore, they cannot be garbled completely independently. In our case, we will assume that the garbling procedure of each program P_i gets t^{init} which is the total number of CPU steps executed so far by $P_1, \ldots, P_{i-1}$ and t^{cur} which is the number of CPU steps to be executed by P_i.

Syntax and Efficiency. A *garbled RAM* scheme consists of four procedures: (GData, GProg, GInput, GEval) with the following syntax:

- $\tilde{D} \leftarrow \mathsf{GData}(D, k)$: Takes memory data $D \in \{0,1\}^n$ and a key k. Outputs the garbled data $\tilde{D}$.
- $(\tilde{P}, k^{\mathsf{in}}) \leftarrow \mathsf{GProg}(P, k, n, t^{init}, t^{cur})$: Takes a key k and a description of a RAM program P with memory-size n and run-time consisting of t^{cur} CPU steps. In the case of garbling multiple programs, we also provide t^{init} indicating the cumulative number of CPU steps executed by all of the previous programs. Outputs a garbled program $\tilde{P}$ and an input-garbling-key k^{in}.
- $\tilde{x} \leftarrow \mathsf{GInput}(x, k^{\mathsf{in}})$: Takes an input x and input-garbling-key k^{in} 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 computes the output $y = P^D(x)$. We model GEval itself as a RAM program that can read and write to arbitrary locations of its memory initially containing $\tilde{D}$.

We require that the run-time of GData be $O(n \cdot \mathsf{poly}(\kappa))$, which also serves as an upper bound on the size of $\tilde{D}$, and also require that the run-time of GInput should be $|x| \cdot \mathsf{poly}(\kappa)$. We also wish to minimize the run-time of GProg and GEval, preferably as low as $\mathsf{poly}(\kappa)\mathsf{polylog}(n) \cdot (|P| + t^{cur})$ for GProg and $\mathsf{poly}(\kappa)\mathsf{polylog}(n) \cdot t^{cur}$ for GEval (but not all our constructions achieve polylogarithmic overhead in n).

Correctness and Security. To define the correctness and security requirements of garbled RAMs, let $P_1, \ldots, P_\ell$ be any sequence of programs with polynomially-bounded run-times $t_1, \ldots, t_\ell$. Let $D \in \{0,1\}^n$ be any initial memory data, let $x_1, \ldots, x_\ell$ be inputs and $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$ be the outputs given by the sequential execution of the programs. Consider the following experiment: choose a key $k \leftarrow \{0,1\}^\kappa$, $\tilde{D} \leftarrow \mathsf{GData}(D, k)$ and for $i = 1, \ldots, \ell$:

$$(\tilde{P}_i, k_i^{\mathsf{in}}) \leftarrow \mathsf{GProg}\left(P_i, n, t_i^{init}, t_i, k\right), \tilde{x}_i \leftarrow \mathsf{GInput}(x_i, k_i^{\mathsf{in}})$$

where $t_i^{init} := \sum_{j=1}^{i-1} t_i$ denotes the run-time of all programs prior to P_i. Let

$$(y'_1, \ldots, y'_\ell) = (\mathsf{GEval}(\tilde{P}_1, \tilde{x}_1), \ldots, \mathsf{GEval}(\tilde{P}_\ell, \tilde{x}_\ell))^{\tilde{D}},$$

denotes the output of evaluating the garbled programs sequentially over the garbled memory. We require that the following properties hold:

- **Correctness:** We require that $\Pr[y'_1 = y_1, \ldots, y'_\ell = y_\ell] = 1$ in the above experiment.
- **Security:** we require that there exists a universal simulator Sim such that:

$$(\tilde{D}, \tilde{P}_1, \ldots, \tilde{P}_\ell, \tilde{x}_1, \ldots, \tilde{x}_\ell) \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\kappa, \{P_i, t_i, y_i\}_{i=1}^\ell, n).$$

Our security definition is non-adaptive: the data/programs/inputs are all chosen ahead of time. This makes our definitions/analysis simpler and also matches the standard definitions for our building blocks such as ORAM. However, there does not seem to be any inherent hurdle to allowing each subsequent program/input (P_i, x_i) to be chosen adaptively after seeing $\tilde{D}, (\tilde{P}_1, \tilde{x}_1), \ldots, (\tilde{P}_{i-1}, \tilde{x}_{i-1})$.

Security with Unprotected Memory Access (UMA). We also consider a weaker security notion, which we call security with *unprotected memory access* (UMA). In this variant, the attacker may learn the initial contents of the memory D, as well as the complete memory-access pattern throughout the computation including the locations being read/written and their contents. In particular, we let $\mathsf{MemAccess} = \{(i_j^{\mathsf{read}}, i_j^{\mathsf{write}}, b_j^{\mathsf{write}}) : j = 1, \ldots, t\}$ correspond to the outputs of the CPU-step circuits during the execution of $P^D(x)$. For security with unprotected memory access, we give the simulator the additional values $(D, \mathsf{MemAccess})$. Using the notation from above, we require:

$$(\tilde{D}, \tilde{P}_1, \ldots, \tilde{P}_\ell, \tilde{x}_1, \ldots, \tilde{x}_\ell) \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\kappa, \{P_i, t_i, y_i\}_{i=1}^\ell, D, \mathsf{MemAccess}, n).$$

In the long version [9], we show a general transformation that converts any garbled RAM scheme with UMA security into one with full security by encrypting the memory contents and applying oblivious RAM to hide the access pattern. Therefore, it is useful to focus on achieving just UMA security.

3 The Original Lu-Ostrovsky Construction

We now describe the main ideas behind the Lu-Ostrovsky construction from Eurocrypt 2013 [14] (but we use a substantially different exposition). In this extended abstract we only consider security with unprotected memory access (UMA), which completely abstracts out the use of oblivious RAM. Moreover, for ease of exposition, we begin by describing a solution for the case of "read-only" computation, which never writes to memory. Many of the main ideas, as well as the circularity problem, are already present in this simple case.

3.1 Garbling Read-Only Programs

Garbled Data. The garbled data $\tilde{D}$ consists of n secret keys for some symmetric-key encryption scheme. For each bit $i \in [n]$ of the original data D, the garbled

data $\tilde{D}$ contains a secret key sk_i. The secret keys are chosen pseudo-randomly using a pseudo-random function (PRF) family F_k via $\mathsf{sk}_i = F_k(i, D[i])$. Note that, given k, there are two possible values $\mathsf{sk}_{(i,0)} = F_k(i, 0)$ and $\mathsf{sk}_{(i,1)} = F_k(i, 1)$ that can reside in $\tilde{D}[i]$ depending on the bit $D[i]$ of the original data, and we set $\tilde{D}[i] = \mathsf{sk}_{(i,D[i])}$.

Garbled Program (Overview). The garbled program P consists of t garbled copies of an "augmented" CPU-step circuit $C^P_{\mathsf{CPU}^+}$, which we describe shortly. Recall that the basic CPU-step circuit takes as input the current CPU state and the last read bit $(\mathtt{state}, b^{\mathsf{read}})$ and outputs $(\mathtt{state}', i^{\mathsf{read}})$ containing the updated state and the next read location – we can ignore the other outputs $i^{\mathsf{write}}, b^{\mathsf{write}}$ since we are considering read-only computation.

We can garble copy j of the CPU-step circuit so that the labels for the output wires corresponding to the output $\mathtt{state}'$ match the labels of the input wires corresponding to the input $\mathtt{state}$ in the next copy $j + 1$ of the circuit. This allows the garbled state to securely travel from one garbled CPU-step circuit to the next. Each garbled copy j of the CPU-step circuit can also output the read location $i = i^{\mathsf{read}}$ in the clear. The question becomes, how can the evaluator incorporate the data from memory into the computation? In particular, let $\mathsf{lbl}_0^{(\mathsf{read},j+1)}, \mathsf{lbl}_1^{(\mathsf{read},j+1)}$ be the labels of the input wires corresponding to the bit b^{read} in garbled copy $j + 1$ of the circuit. We need to ensure that the evaluator who knows $\mathsf{sk}_{(i,b)} = F_k(i, b)$ can learn $\mathsf{lbl}_b^{(\mathsf{read},j+1)}$ but learns nothing about the other label. Unfortunately, the labels $\mathsf{lbl}_b^{(\mathsf{read},j+1)}$ need to be created at "compile time" when the garbled program is created, and therefore cannot depend on the location $i = i^{\mathsf{read}}$ which is only known at "run time" when the garbled program is being evaluated. Therefore the labels $\mathsf{lbl}_b^{(\mathsf{read},j+1)}$ cannot depend on the keys $\mathsf{sk}_{(i,b)}$ since i is not known.

Lu and Ostrovsky propose a clever solution to the above problem. We *augment* the CPU-step circuit so that the jth copy of the circuit outputs a *translation mapping* $\mathsf{translate}$ which allows the evaluator to translate between the keys $\mathsf{sk}_{(i,b)}$ contained in the garbled memory and the labels $\mathsf{lbl}_b^{(\mathsf{read},j+1)}$ of the read-bit in the next circuit. The translation mapping is computed by the jth CPU circuit at *run-time* and therefore can depend on the memory location $i = i^{\mathsf{read}}$ being requested in that step. The translation mapping computed by circuit j consists of two ciphertexts $\mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1)$ where ct_b is an encryption of the label $\mathsf{lbl}_b^{(\mathsf{read},j+1)}$ under the secret key $\mathsf{sk}_{(i,b)} = F_k(i, b)$.[4] In order to compute this encryption, the augmented CPU-step circuits contain the PRF key k as a *hard-coded value.*

Garbled Program (Technical). In more detail, we define an augmented CPU-step circuit $C^P_{\mathsf{CPU}^+}$ which gets as input $(\mathtt{state}, b^{\mathsf{read}})$ and outputs $(\mathtt{state}', i^{\mathsf{read}},$

[4] Since we are only aiming for UMA security, we can reveal the bit b and therefore do not need to permute the ciphertexts.

$\mathsf{translate}$). It contains some hard-coded parameters $(k, r_0, r_1, \mathsf{lbl}_0^{(\mathsf{read})}, \mathsf{lbl}_1^{(\mathsf{read})})$ and performs the following computation:

- $(\mathtt{state}', i^{\mathsf{read}}) = C^P_{\mathsf{CPU}}(\mathtt{state}, b^{\mathsf{read}})$ are the outputs of the basic CPU-step circuit.
- $\mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1)$ consists of two ciphertexts, computed as follows. For $b \in \{0,1\}$, first compute $\mathsf{sk}_{(i,b)} := F_k(i,b)$ for $i = i^{\mathsf{read}}$. Then set $c_b = \mathsf{Enc}_{\mathsf{sk}_{(i,b)}}(\mathsf{lbl}_b^{(\mathsf{read})}; r_b)$ where Enc is a symmetric key encryption and r_b is the encryption randomness.

The garbled program $\tilde{P}$ consists of t garbled copies of this augmented CPU-step circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j)$. We start garbling from the end $j = t$. Each garbled circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j)$ outputs the values $i^{\mathsf{read}}, \mathsf{translate}$ in the clear and the updated $\mathtt{state}'$ is garbled with the same labels as the input $\mathtt{state}$ in the next circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j+1)$; the last circuit outputs $\mathtt{state}'$ in the clear as the output of the computation. Each garbled circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j)$ contains hard-coded values $(k, r_0^{(j)}, r_1^{(j)}, \mathsf{lbl}_0^{(\mathsf{read},j+1)}, \mathsf{lbl}_1^{(\mathsf{read},j+1)})$ which are used to compute the translation mapping $\mathsf{translate}$ as described above. The key k is the PRF key which was used to garbled the memory data. The values $r_0^{(j)}, r_1^{(j)}$ are fresh encryption random coins, and $\mathsf{lbl}_0^{(\mathsf{read},j+1)}, \mathsf{lbl}_1^{(\mathsf{read},j+1)}$ are the labels of the input-wire for the bit b^{read} in the garbled circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j+1)$.

Garbled Input and Evaluation. The garbled input $\tilde{x}$ consists of the wire-labels for the value $\mathtt{state}_1 = x$ for the garbled circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j=1)$. The evaluator simply evaluates the garbled augmented CPU-step circuits one by one starting from $j = 1$. It can evaluate the first circuit using only $\tilde{x}$, and gets out a garbled output $\mathtt{state}_2$ along with the values $(i^{\mathsf{read}}, \mathsf{translate} = (c_0, c_1))$ in the clear. The evaluator looks up the secret key $\mathsf{sk} := \tilde{D}[i^{\mathsf{read}}]$ and attempts to use it to decrypt c_0 and c_1 to recover a label $\mathsf{lbl}^{(\mathsf{read},j=2)}$. The evaluator then evaluates the second garbled circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j=2)$ using the garbled input $\mathtt{state}_2$ and the wire-label $\mathsf{lbl}^{(\mathsf{read},j=2)}$ for the wire corresponding to the bit b^{read}. This process continues until the last circuit $j = t$ which outputs $\mathtt{state}'$ in the clear as the output of the computation.

3.2 Circularity in the Security Analysis

There is good intuition that the above construction *should* be secure. In particular, the evaluator only gets one label per wire of the first garbled circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j=1)$ and therefore does not learn anything beyond its outputs $i = i^{\mathsf{read}}, \mathsf{translate}$ (in the clear) and the garbled value $\mathtt{state}_2$ which can be used as an input to the second circuit. Now, assume that the memory-data contains (say) the bit $D[i] = 0$ and so the evaluator can get $\mathsf{sk}^{(i,0)}$ from the garbled memory $\tilde{D}$. Using the translation map $\mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1)$, the evaluator can use this to recover the label $\mathsf{lbl}_0^{\mathsf{read}}$ corresponding to the read-bit $b^{\mathsf{read}} = 0$ of the next circuit $j = 2$. We need to argue that the evaluator does not learn anything about the

"other" label: $\mathsf{lbl}_1^{\mathsf{read}}$. Intuitively, the above should hold since the evaluator does not have the secret key $\mathsf{sk}_{(i,1)} = F_k(i,1)$ needed to decrypt ct_1. Unfortunately, in attempting to make the above intuition formal, we uncover a complex circularity:

1. In order to argue that the evaluator does not learn anything about the "other" label $\mathsf{lbl}_1^{\mathsf{read}}$, we need to rely on the security of the ciphertext ct_1.
2. In order to rely on the security of the ciphertext ct_1 we need to argue that the attacker does not learn the decryption key $\mathsf{sk}_{(i,1)} = F_k(i,1)$, which requires us to argue that the attacker does not learn the PRF key k. However, the PRF key k is contained as a hard-coded value of the second garbled circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j=2)$ and all future circuits as well. Therefore, to argue that the attacker does not learn k we need to (at the very least) rely on the security of the second garbled circuit.
3. In order to use the security of the second garbled circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j=2)$, we need to argue that the evaluator only gets one label per wire, and in particular, we need to argue the the evaluator does not have the "other" label $\mathsf{lbl}_1^{\mathsf{read}}$. But this is what we wanted to prove in the first place!

We note that the above can be seen as a complex circularity problem involving the PRF, the encryption scheme and the garbled circuit. In particular, the PRF key k is used to encrypt both labels for some input-wire in the garbled circuit, but k is also a hard-coded in the garbled circuit. Therefore we cannot rely on the security of the garbled circuit unless we argue that k stays hidden, but we cannot argue that k stays hidden without relying on the security of the garbled circuit. Notice that this circularity problem comes up even if the evaluator didn't get the garbled data $\tilde{D}$ at all.

The problem is even more complex than described above since the key k is hard-coded in many other garbled circuits and the outputs of these circuits depend on k but do not reveal k directly. Therefore, the circularity problem is not "contained" to a single circuit. We do not know of any "simple" circular-security assumption that one could make on the circuit-garbling scheme, the PRF, and/or the encryption scheme that would allow us to prove security, other than simply assuming that the full construction is secure.

3.3 Writing to Memory

We now describe the main ideas behind how to handle "writes" in the Lu-Ostrovsky construction. Although the circularity problem remains in this solution, it will be useful to see the ideas as they will guide us in our fixes. We again note that our exposition here is substantially different from [14].

Predictably Timed Writes. Below we describe how to incorporate a limited form of writing to memory, which we call *predictably timed writes* (ptWrites). On a high level, this means that whenever we want to *read* some location i in memory, it is easy to figure out the time (i.e., CPU step) j in which that location was last *written* to, given only the current state of the computation and without reading

any other values in memory. In the long version [9] we describe how to upgrade a solution for ptWrites to one that allows arbitrary writes. We give a formal definition of ptWrites below:

Definition 1 (Predictably Timed Writes (ptWrites)). *A program execution $P^D(x)$ has predictably timed writes (ptWrites) if there exists a poly-size circuit* $\mathsf{WriteTime}$ *such that the following holds for every CPU step $j = 1, \ldots, t$. Let the inputs/outputs of the jth CPU step be $C^P_{\mathsf{CPU}}(\mathtt{state}_j, b_j^{\mathsf{read}}) = (\mathtt{state}_{j+1}, i_j^{\mathsf{read}}, i_j^{\mathsf{write}}, b_j^{\mathsf{write}})$. Then, $u = \mathsf{WriteTime}(j, \mathtt{state}_j, i_j^{\mathsf{read}})$ is the largest value of $u < j$ such that the CPU step u wrote to location i_j^{read}; i.e., $i_u^{\mathsf{write}} = i_j^{\mathsf{read}}$. We also define a ptWrites property for a sequence of program executions $(P_1(x_1), \ldots, P_\ell(x_\ell))^D$ if the above property holds for each CPU step in the sequence.*

Garbling programs with ptWrites. At any point in time, the garbled memory data $\tilde{D}$ maintained by the honest evaluator should consist of secret keys of the form $\mathsf{sk}_{(j,i,b)} = F_k(j, i, b)$ for each location $i \in [n]$, where the additional value j will denote a "time step" in which the location i was last written to, and b denotes the current bit in that location. Initially, for each location $i \in [n]$, we set $\tilde{D}[i] = \mathsf{sk}_{(0,i,D[i])}$ using the time period $j = 0$. Then, to read from a location i with last-write-time u, the CPU circuit encrypts the wire-label for bit b under some key which depends on (u, i, b), and to write a bit b to location i in time-step j the CPU circuit gives out some key which depends on (j, i, b).

In more detail, to write a bit b to memory location i^{write} in time step u, the augmented CPU circuit now simply computes a secret key $\mathsf{sk}_{(u,i,b)} = F_k(u, i, b)$, using the hard-coded PRF key k, and outputs $\mathsf{sk}_{(u,i,b)}$ in the clear. The honest evaluator will place this new key in to garbled memory by setting $\tilde{D}[i] := \mathsf{sk}_{(u,i,b)}$, and can "forget" the previous key in location i.

To read from location i^{read}, in time step j we now need to make sure that the evaluator can *only* use latest key (corresponding to the most recently written bit), and cannot use some outdated key (corresponding to an old value in that location). To do so, the augmented CPU circuit computes the last write time for the location i^{read} by calling $u = \mathsf{WriteTime}(j, \mathtt{state}_j, i^{\mathsf{read}})$ and then prepares the translation mapping $\mathsf{translate} = (c_0, c_1)$ as before, but with respect to the keys for time step u by encrypting the ciphertext c_0, c_1 under the secret key $\mathsf{sk}_{(u,i,0)} = F_k(u, i, 0), \mathsf{sk}_{(u,i,1)} = F_k(u, i, 1)$ respectively.

4 Our Solution Using IBE

We now outline our modifications to the Lu-Ostrovsky solution so as to remove the circular use of garbled circuits, using identity-based encryption. See the long version [9] for a full description. As above, we begin by describing our fix for read-only computation and then describe how to handle ptWrites.

4.1 A Read-Only Construction

The initial idea is to simply replace the symmetric-key encryption scheme with a public-key one. Each garbled circuit will have a hard-coded public-key which allows it to create ciphertexts $\mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1)$, but does not provide enough information to "break" the security of these ciphertexts. Unfortunately, standard public-key encryption does not suffice and we will need to rely on *identity-based encryption* (IBE). Indeed, we can already think of the Lu-Ostrovsky construction outlined above as implicitly using a "symmetric-key" IBE where the master secret key k is needed to encrypt. In particular, we can think of the garbled memory data as consisting of "identity secret keys" $sk_{(i,b)}$ for identities of the form $(i, b) \in [n] \times \{0, 1\}$ depending on the data bit $b = D[i]$. The translation information consists of an encryption of the label $\mathsf{lbl}_0^{\mathsf{read}}$ for identity $(i, 0)$ and an encryption of $\mathsf{lbl}_1^{\mathsf{read}}$ for identity $(i, 1)$. We can view the Lu-Ostrovsky scheme as using a symmetric-key IBE scheme constructed from a PRF $F_k(\cdot)$ and a standard encryption scheme, where the encryption of a message msg for identity id is computed as $\mathsf{Enc}_{F_k(\mathsf{id})}(\mathsf{msg})$. We now simply replace this with a public-key IBE. In particular, we modify the augmented CPU-step circuit so that it now contains a hard-coded master public key MPK for an IBE scheme (instead of a PRF key k) and it now creates the translation map $\mathsf{translate} = (c_0, c_1)$ by setting $c_b = \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id} = (i, b), \mathsf{msg} = \mathsf{lbl}_b^{(\mathsf{read})})$ to be an encryption of the message $\mathsf{lbl}_b^{\mathsf{read}}$ for identity (i, b).

Overview of Security Proof. The above scheme already removes the circularity problem and yields a secure construction for read-only computation with unprotected memory-access (UMA) security. In particular, we can now rely on the semantic-security of the IBE ciphertexts created by a garbled circuit j without needing to argue about the security of future garbled circuits $j + 1, j + 2, \ldots$ since they do not contain any secret information about the IBE scheme.

4.2 Writing to Memory

We present the solution for a predictably timed writes (ptWrites), cf. Definition 1. To handle writes, we now want the garbled data to consist of secret keys for identities of the form $\mathsf{id} = (j, i, b)$ where $i \in [n]$ is the location in the data, j is a time step when that location was last written to, and $b \in \{0, 1\}$ is the bit that was written to location i in time j. The honest evaluator only needs to keep the the most recent secret key for each location i. When the computation needs to read from location i, it computes the last time step j when this location was written to, then creates the translation mapping by encrypting ciphertexts for the two identities (j, i, b) for $b = 0, 1$.

When the computation needs to write a bit b to location i in time period j, the corresponding garbled circuit should output a secret key for the identity (j, i, b). Unfortunately, a naive implementation would require the garbled circuits to include the master secret key MSK of the IBE in order to compute these secret keys, and this would re-introduce the same circularity problem that we are trying

to avoid! Instead we use a solution similar to hierarchical IBE (HIBE), as we describe next.

Timed IBE. To avoid circularity, we introduce a primitive that we call a *timed* IBE (TIBE) scheme. Such a scheme roughly lets us create "time-period keys" TSK_j for arbitrary time periods $j \geq 0$ such that TSK_j can be used to create identity-secret-keys $\mathsf{sk}_{(j,v)}$ for arbitrary v, but cannot break the security of any other identities with $j' \neq j$.[5] TIBEs as described above can be easily constructed from 2-level HIBE by thinking of the identities (j, v) as being of the form $j.v$ where the time-period j is the top level of the hierarchy and v is the lower level; the time-period key TSK_j would just be a secret key for the identity j. We note, however, that for our purposes we can use a slightly weaker version of TIBEs where we only give out limited number of keys, and these can be constructed from any selectively-secure IBE scheme.

Definition 2 (Timed IBE (TIBE)). *A TIBE scheme Consists of 5 PPT algorithms* $\mathsf{MasterGen}$, $\mathsf{TimeGen}$, KeyGen, Enc, Dec *with the syntax:*

- $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MasterGen}(1^\kappa)$*: generates master public/secret keys* MPK, MSK.
- $\mathsf{TSK}_j \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, j)$*: Generates a key for time-period* $j \in \mathbb{N}$.
- $\mathsf{sk}_{(j,v)} \leftarrow \mathsf{KeyGen}(\mathsf{TSK}_j, (j, v))$*: creates a secret key for the identity* (j, v).
- $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((j, v), \mathsf{msg})$ *encrypts* msg *for the identity* (j, v).
- $\mathsf{msg} = \mathsf{Dec}_{\mathsf{sk}_{(j,v)}}(\mathsf{ct})$*: decrypts* ct *for identity* (j, v) *using the secret key* $\mathsf{sk}_{(j,v)}$.

The scheme should satisfy the following properties:
Correctness: *For any* $\mathsf{id} = (j, v)$*, and any* $\mathsf{msg} \in \{0,1\}^*$ *it holds that:*

$$\Pr\left[\mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}) = \mathsf{msg} \;\middle|\; \begin{array}{c}(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MasterGen}(1^\kappa), \mathsf{TSK}_j \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, j), \\ \mathsf{sk} \leftarrow \mathsf{KeyGen}(\mathsf{TSK}_j, (j, v)), \mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((j, v), \mathsf{msg})\end{array}\right] = 1.$$

Security: *Consider the following game between an attacker* $\mathcal{A}$ *and a challenger.*

- *The attacker* $\mathcal{A}(1^\kappa)$ *chooses target identity* $\mathsf{id}^* = (j^*, v^*)$ *and bound* $t \geq j^*$ *(given in unary). The attacker also chooses a set of identities* $S = S_0 \cup S_{>0}$ *with* $\mathsf{id}^* \notin S$ *such that: (I)* S_0 *contains arbitrary identities of the form* $(0, v)$*, (II)* $S_{>0}$ *contains exactly one identity* (j, v) *for each period* $j \in \{1, \ldots, j^*\}$*. Lastly, the adversary chooses messages* $\mathsf{msg}_0, \mathsf{msg}_1 \in \{0,1\}^*$ *of equal size* $|\mathsf{msg}_0| = |\mathsf{msg}_1|$.
- *The challenger chooses* $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MasterGen}(1^\kappa)$*, and* $\mathsf{TSK}_j \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, j)$ *for* $j = 0, \ldots, t$*. For each* $\mathsf{id} = (j, v) \in S$ *it chooses* $\mathsf{sk}_{\mathsf{id}} \leftarrow \mathsf{KeyGen}(\mathsf{TSK}_j, \mathsf{id})$*. Lastly, the challenger chooses a challenge bit* $b \leftarrow \{0,1\}$ *and sets* $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id}^*, \mathsf{msg}_b)$*. The challenger gives the attacker:*

$$\mathsf{MPK} \quad , \quad \overline{\mathsf{TSK}} = \{\mathsf{TSK}_j\}_{j^* < j \leq t} \quad , \quad \overline{\mathsf{sk}} = \{(\mathsf{id}, \mathsf{sk}_{\mathsf{id}})\}_{\mathsf{id} \in S} \quad , \quad \mathsf{ct}.$$

- *The attacker outputs a bit* $\hat{b} \in \{0,1\}$.

[5] In our use of TIBE, we will always set $v = (i, b)$ for some $i \in [n], b \in \{0,1\}$.

The scheme is secure if, for all PPT $\mathcal{A}$, we have $|\Pr[b = \hat{b}] - \frac{1}{2}| \leq \mathsf{negl}(\kappa)$ *in the above game.*

In the full version [9], we show how to construct a TIBE scheme from *any* secure IBE scheme.

Solution Using TIBE. Using a TIBE scheme, we can solve the problem of writes. For each location $i \in [n]$ the honest evaluator will always have a secret key for identity $\mathsf{id} = (j, i, b)$ where j is the last-write-time for location i and $b \in \{0, 1\}$ is its value. Initially, the garbled data consists of secret keys for the time period $j = 0$. Each augmented-CPU-step-circuit in time period $j > 0$ will contain a hard-coded time-period key TSK_j and the master-public-key MPK. This allows each CPU step j to read an arbitrary location $i \in [n]$ with last-write time $u < j$ by encrypting the translation ciphertexts $\mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1)$ under MPK to the identities (u, i, b) for $b = 0, 1$. Each such CPU step j can also write a bit b to an arbitrary location i by creating a secret key $\mathsf{sk}_{\mathsf{id}}$ for the identity $\mathsf{id} = (j, i, b)$ using TSK_j. Notice that we create at most one such secret-key for each time period $j > 0$. This solution does not suffer from a circularity problem, since the ciphertexts created by CPU step j for an identity (u, i, b) must have $u < j$, and therefore we can rely on semantic security even given the hard-coded values $\mathsf{TSK}_{j+1}, \ldots, \mathsf{TSK}_t$ in all future garbled circuits.

5 Our Solution Using One-Way Functions

The main problem that arises in the circularity is that there is only one PRF key, and that this key when embedded in any future time step is able to decode anything the circuit does in the current time step. The intuitive way to circumvent this is to iteratively weaken the PRF key. In order to do so, we introduce the following notion of revocable PRFs.

5.1 Revocable PRFs

We define the notion of (adaptively) revocable PRFs and we explain how it differs from existing notions such as [6,7,13,18]. The idea is that we can revoke values from the key so that the PRF cannot be evaluated on these values, and given an already-revoked key, one can further revoke new values.

Definition 3. *A* revocable *PRF is a PRF F equipped with an additional revoke algorithm* Rev. *The keys for this PRF are of the form k_X where X is a subset of the domain (which is the revoked values), and we identify "fresh keys" with $k_\emptyset$. The revoke algorithm takes as input a key k_X and another subset Y, and output $k_{X \cup Y}$, satisfying the following properties:*

Correctness: *$F_{k_{X \cup Y}}(x) = F_{k_X}(x)$ if $x \notin X \cup Y$, and $F_{k_{X \cup Y}}(x) = \perp$ otherwise.*
Pseudorandomness: *Given any set of keys $\{k_{Y_1}, \ldots, k_{Y_m}\}$, $F_k(x)$ is pseudorandom for all $x \notin \bigcap_i Y_i$.*

Note that this definition appears similar to constrained PRFs [6]; however, we do not require that the revoked set to be hidden in any way, and we allow successive revocation of more and more values starting from an initial fresh key.

Revocable PRFs can be constructed based on the GGM construction [10], as we now sketch. Recall that GGM PRF is built out of a length-doubling PRG G. It can be thought of as filling the nodes of a complete binary tree with pseudo-random values: The PRF key is placed in the root, and then the values in all other nodes are computed by taking any node with value s and putting in its two children the two halves of the pseudo-random value $G(s)$. An input to the PRF specifies a leaf in the tree, and the corresponding output is the pseudo-random value in that leaf. To revoke a single leaf x, we simply replace the root value with the values of all the siblings of nods on the path to the revoked leaf. Clearly we can still compute the PRF on every input $y \neq x$, but the value of x is now pseudo-random even given the weaker key. More generally, let $X = \{x_1, x_2, \ldots, x_s\}$ be a set of s leaves that we want to revoke, the key k_X will contain siblings of nodes on the paths to all these x_i's, except these nodes that are themselves ancestors of some x_i. This key consists of at most $s \log N$ values, where N is the number of leaves in the tree.

5.2 Overview of the Second Construction

Step 1, read-once programs. We begin by describing a naive construction that solves the circularity issue by using revocable PRFs. Starting from a RAM program with ptWrites, we convert it to a "read-once" program (i.e. no location is read more than once before it is overwritten) by introducing a local cache in which the CPU keeps every value that it gets from memory. Of course this transformation comes with a steep performance price, as the CPU state after t steps grows to size roughly $\min(n, t)$.

Once we have a read-once program, we can use revocable PRFs to break the circularity in the original Lu-Ostrovsky construction as follows: instead of having the PRF key hard-wired in the augmented CPU-step circuit, we now let it be part of the input. In particular the circuit will get some key K_X as part of the input, compute the next address i to read from memory (if any), and will prepare the translation mapping table $\mathsf{translate}$ by evaluating $F_{K_X}(u, i, 0)$ and $F_{K_X}(u, i, 1)$, then revoke the two points $(u, i, 0)$ and $(u, i, 1)$, and pass to the next CPU-step function the PRF key $K_{X'}$ for $X' = X \cup \{(i, 0), (i, 1)\}$. Since this is a read-once program, then no future CPU step ever needs to evaluate the PRF at these points. Writing remains unchanged, and does not interfere with the PRF because all revocations only happened to CPU-steps in the past, and in the proof we can still plant the PRF challenge in the un-written bit $F_{K_X}(u, i, 1 - b)$.

This solves the circularity problem since the encryption in $\mathsf{translate}$ is secure even given all future keys K'_x. Moreover, the size of the augmented CPU step is not much larger than that of the original CPU step since the keys K_X only grow to size roughly $\kappa \cdot \min(n, t) \cdot \log n$. The naive construction would just garble all of these enhanced CPU steps, which entails time complexity of $\mathsf{poly}(\kappa) \cdot \min(n, t) \cdot t \log n$ for both GProg and GEval. This solution, although secure, is not much

better than just converting the original RAM program to a circuit and garbling that circuit (i.e. the generic t^3 transformation). We do obtain some *amortized* savings in the case of running multiple programs on persistent memory (e.g. repeated binary search).

We mention that instead of "read-once", we can consider a weakened version that allows for some bounded number of reads before a location is overwritten. In such a case, there exists a transformation (via ORAM, see [15]) from an arbitrary RAM program to one that satisfies ptWrites and poly-logarithmic bounded reads, *without a cache*. In order for GRAM to handle multiple reads to the same location, we can simply apply repetition and have multiple, independent PRFs. Unfortunately, even though this removes the cache, the bottleneck remains in the growth of the keys K_X. Only when combined with a more efficient revocable PRF scheme will this result in lower overhead. Instead, we propose the following approach.

Step 2, refreshing the memory. To reduce the complexity, we introduce a periodic memory-refresh operation which is designed to rein-in the growth in the (augmented) CPU-step functions: Namely, we refresh the entire memory and its representation every f steps, for some parameter $f < n$ to be determined later. In more detail, after every f CPU steps we introduce an special refresh circuit that (a) empties the cache, and (b) re-garbles the memory using a freshly chosen PRF key. The complexity of each such refresh step is $\mathsf{poly}(k) \cdot n$, and there is no need to hard-wire in it any PRF keys (since we can instead just hard-wire all the $O(n)$ PRF values that it needs, rather than computing them.)

The advantage of using these refresh steps is that now the augmented CPU steps only grow up to size at most $O(\kappa \cdot f \log n)$, and although each refresh step is expensive we only have t/f such steps. Hence the overall complexity is bounded by $\mathsf{poly}(\kappa) \cdot (t/f \cdot n + t \cdot f \log n)$. Setting $f = \sqrt{n/\log n}$ thus yields overall complexity of $\mathsf{poly}(\kappa) \cdot t \cdot \sqrt{n \log n}$ for performing t steps, so we get overhead of $\mathsf{poly}(\kappa)\sqrt{n \log n}$.

Step 3, a recursive construction. To further reduce the overhead, we notice that instead of garbling the augmented CPU steps as circuits (which incurs complexity $s \cdot \mathsf{poly}(\kappa)$ for a size-s step), we can instead view these steps as RAM programs and apply the same RAM-garbling procedure recursively to these programs. Each augmented CPU state grows to $O(\kappa \cdot f \log n)$, and can be implemented as a RAM program of that size, but running in time $\tilde{O}(\kappa \log f \log n)$ by using appropriate data structures. This allows us to balance the refresh time with the cost of executing each of the t emulated steps as a recurrence relation. There are additional details required when applying this recursion. Since every level of the recursion induces a factor of κ, we must choose f so that the savings overcome this factor and while preserving polynomial complexity. Also, if we treat each step as an independent GRAM, the cost of running GData on the size-f cache would negate all savings. We must amortize this cost by treating the steps as running on the same persistent "mini-GRAM" memory. This requires a careful formalization of our recursion in terms of a composition theorem that states that a small, secure

GRAM can be bootstrapped into a larger one via treating CPU internals as persistent memory. In the long version [15] we give the details and show that for any constant $\varepsilon > 0$ we can choose the parameter f and the number of recursion levels to get overhead of $\mathsf{poly}(\kappa)n^{\varepsilon}$.

6 Conclusions

We conclude with two important open problems. Firstly, it would be interesting to give a garbled RAM scheme based only on the existence of one-way functions with poly-logarithmic overhead. Secondly, the work of Goldwasser et al. recently constructed the first *reusable* garbling schemes for *circuits* and *Turing machines* [12,11] where the garbled circuit/TM can be executed on multiple inputs. It would be interesting to analogously construct a reusable garbled RAM where the garbled program can be evaluated on many different "short" inputs.

Acknowledgments. The first two authors were supported in part by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20202. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

The third author was supported in part by NSF grants CCF-0916574; IIS-1065276; CCF-1016540; CNS-1118126; CNS-1136174.

The fourth author was supported in part by NSF grants CCF-0916574; IIS-1065276; CCF-1016540; CNS-1118126; CNS-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 also 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.

The sixth author was supported in part by NSF grant 1347350.

References

1. Applebaum, B., Ishai, Y., Kushilevitz, E.: From secrecy to soundness: Efficient verification via secure computation. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 152–163. Springer, Heidelberg (2010)
2. Bellare, M., Dowsley, R., Waters, B., Yilek, S.: Standard security does not imply security against selective-opening. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 645–662. Springer, Heidelberg (2012)

3. Bellare, M., Hoang, V.T., Rogaway, P.: Adaptively secure garbling with applications to one-time programs and secure outsourcing. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 134–153. Springer, Heidelberg (2012)
4. Bellare, M., Hofheinz, D., Yilek, S.: Possibility and impossibility results for encryption and commitment secure under selective opening. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 1–35. Springer, Heidelberg (2009)
5. Black, J., Rogaway, P., Shrimpton, T.: Encryption-scheme security in the presence of key-dependent messages. In: Nyberg, K., Heys, H.M. (eds.) SAC 2002. LNCS, vol. 2595, pp. 62–75. Springer, Heidelberg (2003)
6. Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 280–300. Springer, Heidelberg (2013)
7. Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. IACR Cryptology ePrint Archive, 2013:401 (2013)
8. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. J. Comput. Syst. Sci. 7(4), 354–375 (1973)
9. Gentry, C., Halevi, S., Raykova, M., Wichs, D.: Garbled RAM revisited, part I. Cryptology ePrint Archive, Report 2014/082 (2014), `http://eprint.iacr.org/`
10. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions (extended abstract). In: FOCS, pp. 464–479. IEEE Computer Society (1984)
11. Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: How to run turing machines on encrypted data. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 536–553. Springer, Heidelberg (2013)
12. Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) STOC, pp. 555–564. ACM (2013)
13. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Sadeghi, A.-R., Gligor, V.D., Yung, M. (eds.) ACM Conference on Computer and Communications Security, pp. 669–684. ACM (2013)
14. Lu, S., Ostrovsky, R.: How to garble RAM programs? In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 719–734. Springer, Heidelberg (2013)
15. Lu, S., Ostrovsky, R.: Garbled RAM revisited, part II. Cryptology ePrint Archive, Report 2014/083 (2014), `http://eprint.iacr.org/`
16. Pippenger, N., Fischer, M.J.: Relations among complexity measures. J. ACM 26(2), 361–381 (1979)
17. Rothblum, R.D.: On the circular security of bit-encryption. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 579–598. Springer, Heidelberg (2013)
18. Sahai, A., Waters, B.: How to use indistinguishability obfuscation: Deniable encryption, and more. IACR Cryptology ePrint Archive, 2013:454 (2013)
19. Yao, A.C.-C.: Protocols for secure computations (extended abstract). In: FOCS, pp. 160–164 (1982)