Garbled RAM Revisited
Part I

Craig Gentry* Shai Halevi* Mariana Raykova † Daniel Wichs‡

February 5, 2014

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 a subtle yet difficult-to-overcome issue with the Lu-Ostrovsky construction, that prevents a proof of security from going through. Specifically, the construction requires a complex "circular" use of Yao garbled circuits and PRFs. As our main result, we show how to remove this circularity and get a provably secure solution using *identity-based encryption (IBE)*. We also abstract out, simplify and generalize the main ideas behind the Lu-Ostrovsky construction, making them easier to understand and analyze.

In a companion work to ours (Part II), Lu and Ostrovsky show an alternative approach to solving the circularity problem.[1] Their approach relies only on the existence of one-way functions, at the price of higher overhead. Specifically, our construction has overhead $\mathsf{poly}(\kappa)\mathsf{polylog}(n)$ (with κ the security parameter and n the data size), while the Lu-Ostrovsky approach can achieve overhead $\mathsf{poly}(\kappa)n^{\varepsilon}$ for any constant $\varepsilon > 0$. It remains as an 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 [Yao82], garbled circuits have found countless applications in cryptography. Perhaps most importantly, they allow for two-party computation with minimal interaction between the parties. On a basic level, garbled circuits allow a user to convert a circuit C into a garbled version $\tilde{C}$, and an input x into a garbled version $\tilde{x}$ in such a way that $\tilde{C}$ can be evaluated on $\tilde{x}$ to reveal the output $C(x)$, but nothing else is revealed. As with most two-party and multi-party 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.

*IBM Research, T.J. Watson. E-mail: cbgentry@us.ibm.com, shaih@alum.mit.edu. 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.

†SRI. E-mail: mariana@cs.columbia.edu. Research conducted in part while at IBM Research.

‡Northeastern University. E-mail: wichs@ccs.neu.edu. Research conducted in part while visiting IBM Research. Supported in part by NSF grant 1347350.

[1] A merged version of the two works appears in Eurocrypt 2014.

Circuits vs. RAMs. The process of 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 $O(T^3 \log T)$ [CR73, PF79]. 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 [LO13] 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. Similar to Yao garbled circuits, garbling x consists of providing a subset of "wire-labels", which can be exchanged using oblivious transfer (OT) in the context of 2-party computation.

Lu and Ostrovsky proposed a construction of garbled RAMs based on the existence of one-way functions. The construction relies on a clever use of Yao garbled circuits and oblivious RAM (ORAM).

A Circularity Problem. The starting point of this work is a subtle yet difficult-to-overcome issue with the Lu-Ostrovsky construction, that prevents a proof of security from going through. On a high level, the construction requires a complex "circular" use of Yao garbled circuits, which lacks provable security. In particular, the construction provides encryptions of *both* labels (corresponding to bits $0, 1$) of some input wire w in a garbled circuit under some secret-key k, but this secret key k is also hard-coded into the description of the circuit. This introduces the following circularity: we cannot rely on the security of the encryption scheme without relying on the security of the garbled circuit to argue that the key k is hidden, and we cannot rely on the security of the garbled circuit without relying on the security of the encryption scheme to argue that the attacker cannot learn *both* wire labels for w. 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 hardness assumptions. One could draw an analogy to other "subtle" difficulties in cryptography such as circular security [BRS02, Rot13], selective-opening security [BHY09, BDWY12], or adaptively-chosen inputs of garbled circuits [BHR12a], 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. As our main result, we give a new construction of garbled RAMs which removes the circularity problem of the Lu-Ostrovsky construction. The construction relies on the existence of identity-based encryption (IBE). The overhead of the scheme, measured as the evaluation time of a garbled programs vs. the original program, is only $\mathsf{poly}(\kappa)\mathsf{polylog}(n)$, where κ is the security parameter and n is the size of the data. Our construction retains the same main ideas and overall structure of the Lu-Ostrovsky construction,

but we abstract out, simplify and generalize these ideas so as to make them easier to understand and analyze. For example, whereas the previous work relied on specific properties of particular oblivious RAM (ORAM) schemes, we completely abstract out the notion of ORAM and use it in a black-box manner.

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 [LO13], but we proceed to define and analyze it formally.

Another Solution. In a companion work to ours, a new result by Lu and Ostrovsky [LO14] show an alternative approach to removing the circularity problem from their original work [LO13]. Their approach relies only on the existence of one-way functions, but it achieves a worse asymptotic efficiency in terms of the overhead of executing a garbled vs. ungarbled program. Recall that the overhead in our work is $\mathsf{poly}(\kappa)\mathsf{polylog}(n)$, where κ is the security parameter and n is the data size. The result of [LO14] shows that, for any constant $\varepsilon > 0$, there is a scheme with overhead $\mathsf{poly}(\kappa)n^{\varepsilon}$. It remains as an open problem to achieve an overhead of $\mathsf{poly}(\kappa)\mathsf{polylog}(n)$ or even $\mathsf{poly}(\kappa)n^{o(1)}$ assuming only the existence of one-way functions.

Organization. Our construction will rely extensively on standard garbled circuits and on oblivious RAM, which we review in Appendix A. We begin by describing our notation for RAM computation in Section 2 and a definition of garbled RAM Section 3. We then give a high-level abstracted/modularized description of the the solution of Lu-Ostrovsky in Section 4, along with an explanation of the "circularity" issue. In Section 5, we present our main result, describing our "fixed" solution using IBE. Finally, in Section 6 we discuss several extensions/applications of garbled RAM and some open problems.

2 RAM Computation

Notation for RAM Computation. Before we describe *garbled* RAM, let us fix a notation for describing standard RAM computation. We will consider a program P that has random-access to a memory of size n which may initially contain some data $D \in \{0,1\}^n$. In addition, the program gets a "short" input x, which we can alternatively think of as the initial state of the program. 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. 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

[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.

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 `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 $b^{\mathsf{read}}_1 := 0$. In each step j, the computation proceeds by running $C^P_{\mathsf{CPU}}(\mathtt{state}_j, b^{\mathsf{read}}_j) = (\mathtt{state}_{j+1}, i^{\mathsf{read}}, i^{\mathsf{write}}, b^{\mathsf{write}})$. We first read the requested location i^{read} by stetting $b^{\mathsf{read}}_{j+1} := D[i^{\mathsf{read}}]$ 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 $i^{\mathsf{write}} = \perp$.

3 Defining Garbled RAM

We will right-away consider a scenario where 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 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 & 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 $\tilde{D}$.

For **efficiency**, we require that the run-time of GProg, and GEval is $|C^P_{\mathsf{CPU}}| \cdot t^{cur} \cdot \mathsf{poly}(\kappa) \cdot \mathsf{polylog}(n)$, 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 $n \cdot \mathsf{poly}(\kappa)$, which also serves as an upper bound on the size of $\tilde{D}$.

Correctness & 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/analsysis 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 seing $\tilde{D}, (\tilde{P}_1, \tilde{x}_1), \ldots, (\tilde{P}_{i-1}, \tilde{x}_{i-1})$.

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 Section D, 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 our initial effort at just achieving UMA security.

4 The Lu-Ostrovsky Construction

We now describe the main ideas behind the Lu-Ostrovsky construction. Our exposition is substantially different from the original, and the scheme we present is significantly abstracted and modularized. This will make it simpler for us to describe the scheme, highlight the circularity problem, and eventually describe our fix. The same circularity problem is also present in the original scheme described by Lu and Ostrovsky, but would be more difficult to present. As a first step, we will only consider security with unprotected memory access (UMA), which completely abstracts out the use of oblivious RAM. Moreover, for ease of exposition, we will begin by describing a solution for the case of "read-only" computation, which only reads but never writes to memory. Many of the main ideas, as well as the circularity problem, are already present in the "read-only" case.

4.1 Read-Only Solution

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 $\tilde{D}[i] = \mathsf{sk}_i\mathsf{sk}(i, D[i])$.

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 $\mathsf{sk}_{(i,b)} = F_k(i, b)$.[3] 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 $(\mathtt{state}', i^{\mathsf{read}}, \mathsf{translate})$. 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

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

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 & 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 and 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.

4.2 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 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, attempting to make the above intuition formal uncovers 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.

4.3 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 eventual fix. We again note that our exposition here is substantially different from that of Lu and Ostrovsky, and we hope that it is more modular, simpler and easier to understand.

Predictably Timed Writes. As a first step, 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. We will later describe how to upgrade a solution for ptWrites to one that allows arbitrary writes. We give a formal definition of ptWrites below:

Definition 4.1 (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 $C^P_{\mathsf{CPU}}(\mathtt{state}_j, b^{\mathsf{read}}_j) = (\mathtt{state}_{j+1}, i^{\mathsf{read}}_j, i^{\mathsf{write}}_j, b^{\mathsf{write}}_j)$. Then, $u = \mathsf{WriteTime}(j, \mathtt{state}_j, i^{\mathsf{read}}_j)$ is the largest value of $u < j$ such that the CPU step u wrote to location i^{read}_j; i.e., $i^{\mathsf{write}}_u = i^{\mathsf{read}}_j$. 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.*

Overview of Solution. We now describe how garble programs with predictably timed writes (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 $j = 0$.

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.

We repeat the main idea behind this construction, which our eventual "fix" will also rely on: 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).

5 Our Solution Using IBE

We now describe our modifications to the Lu-Ostrovsky solution so as to remove the circular use of garbled circuits. Our solution relies on the use of identity-based encryption. As above, we begin by describing our fix for read-only computation and then describe how to handle ptWrites.

5.1 A Read-Only Construction

Overview of Our Fix. Our 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 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,

5.2 Writing to Memory

We now proceed to describe our complete solution which allows writes to memory. As in the original construction, we first give a solution for predictably timed writes (ptWrites) – see Definition 4.1. Continuing with our general approach of modifying the Lu-Ostrovsky construction to use public-key IBE instead of symmetric-key IBE, we can 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 written to, and $b \in \{0, 1\}$ is the bit that was written to location i in time step j. The honest evaluator only needs to keep the the most recent secret key for each location i. When the computation needs to read a location i, it computes the time step j in which that location was last written to (using the circuit $\mathsf{WriteTime}$) and then creates the translation mapping by encrypting ciphertexts for the identities $\mathsf{id} = (j, i, b)$ for $b = 0, 1$. When the computation needs to write to location i in time period j with some value b, the garbled circuit should simply output a secret key for the identity (j, i, b). Unfortunately, a naive implementation of this solution would require the garbled circuits to have the master secret key MSK of the IBE hard-coded within them in order to compute these secret keys, and this would re-introduce the same circularity problem that we are trying to avoid!

Timed IBE. We solve the above issue by introducing a primitive which we call a *timed* IBE (TIBE) scheme. On a very high level, such a scheme allows us to 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 identities of the form (j, v) for arbitrary v but cannot break the security of any other identities with $j' \neq j$.[4] As a first step, this can be easily accomplished with 2-level hierarchical IBE (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. However, we proceed to define a more careful and restricted notion of TIBE, and show how to construct it from any selectively-secure IBE scheme.[5] Informally, the restricted security property of TIBE says that:

[4] In our use of TIBE, we will always set $v = (i, b)$ for some $i \in [n], b \in \{0, 1\}$ and we will denote the identities $(j, (i, b))$ as (j, i, b) to simplify notation.

[5] The "hurried" reader may skip this definition and proceed to interpret TIBE as a special case of HIBE without it affecting the understanding of the rest of the paper.

- For period $j = 0$, we will give out arbitrarily many secret-keys for identities $(0, v)$, but for each other period $j > 1$ we will give out at most 1 key for an identity (j, v).
- Given "future" time-period keys $\mathsf{TSK}_{j^*+1}, \mathsf{TSK}_{j^*+2}, \ldots, \mathsf{TSK}_t$, and "past" identity keys $\{\mathsf{sk}_{(j,v)}\}$ with $j < j^*$ satisfying the above, semantic security should hold for any identity $\mathsf{id}^* = (j^*, v^*)$ for which a key was not given.

Definition 5.1 (Timed IBE (TIBE))**.** *A TIBE scheme Consists of 5 PPT algorithms* MasterGen, TimeGen, KeyGen, Enc, Dec *with the syntax:*

- $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MasterGen}(1^\kappa)$*: generates master public/secret key pair* MPK, MSK*.*
- $\mathsf{TSK}_j \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, j)$*: Generates a time-period 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)*.*
- $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((j, v), \mathsf{msg})$ *creates an encryption of* msg *under the identity* (j, v)*.*
- $\mathsf{msg} = \mathsf{Dec}_{\mathsf{sk}_{(j,v)}}(\mathsf{ct})$*: decrypts a ciphertexts* ct *for the identity* (j, v) *using a 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: *We consider the following game between an attacker* $\mathcal{A}$ *and a challenger.*

- *The attacker* $\mathcal{A}(1^\kappa)$ *chooses some identity* $\mathsf{id}^* = (j^*, v^*)$ *with* $j^* \in \mathbb{N}$ *and some bound* $t \geq j^*$ *(given in unary). The attacker also chooses a set of identities* $S = S_0 \cup S_{>0}$ *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* $j \in \{1, \ldots, j^*\}$*, (III)* $\mathsf{id}^* \notin S$*. 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* $\hat{b} \in \{0,1\}$*.*

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 Appendix B, we show how to construct a TIBE scheme from *any* IBE scheme. In fact, we only need an IBE scheme with a weak form of selective security.

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.

5.3 Detailed Description

Using the above high-level description, we now give a careful and detailed description of our garbled RAM scheme for programs with ptWrites (predictably timed writes) and satisfying UMA (unprotected memory access) security. Let $(\mathsf{MasterGen}, \mathsf{TimeGen}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a TIBE (see Definition 5.1) scheme. Without loss of generality, we will assume that each secret key $\mathsf{sk}_{\mathsf{id}}$ reveals the identity id that it is intended for. Let $(\mathsf{GCircuit}, \mathsf{CircEval})$ be a circuit garbling scheme with wire labels (see Appendix A.1). We let the key k of the garbled program scheme corresponded to the randomness of the $\mathsf{MasterGen}$ procedure and let $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MasterGen}(1^\kappa; k)$. Since $\mathsf{GData}, \mathsf{GProg}$ both get k as an input, we can assume that they also have $(\mathsf{MPK}, \mathsf{MSK})$.

Garbled Data. The garbled data $\tilde{D}$ consists of n secret keys.

- $\tilde{D} \leftarrow \mathsf{GData}(D, k)$: Compute $\mathsf{TSK}_0 \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, 0)$. For each $i \in [n]$, set $\tilde{D}[i] := \mathsf{sk}_{\mathsf{id}}$ where $\mathsf{sk}_{\mathsf{id}} \leftarrow \mathsf{KeyGen}(\mathsf{TSK}_0, (0, i, D[i]))$ is the secret key for the identity $\mathsf{id} = (0, i, D[i])$.

Notice that the key in location i will now completely reveal $D[i]$ since it reveals its identity id. This is fine since we are (for now) only considering UMA security, where the memory data D need not protected.

Garbled Program. Firstly, we describe the augmented-CPU-step circuit $C^P_{\mathsf{CPU}^+}$ for the program P in Figure 1. The garbled RAM program for P will consist of t copies of a garbled augmented-CPU-Step

Input: $(\mathtt{state}, b^{\mathsf{read}})$ **Output:** $(\mathtt{state}', i^{\mathsf{read}}, i^{\mathsf{write}}, \mathsf{sk}^{\mathsf{write}}, \mathsf{translate})$

Hard-Coded Parameters: $\mathsf{MPK}, \mathsf{TSK}_j, j, r_0, r_1, r_{\mathsf{write}}, \mathsf{lbl}_0^{\mathsf{read}}, \mathsf{lbl}_1^{\mathsf{read}}$

The circuit $C^P_{\mathsf{CPU}^+}$ performs the following computation:

- $(\mathtt{state}', i^{\mathsf{read}}, i^{\mathsf{write}}, b^{\mathsf{write}}) := C^P_{\mathsf{CPU}}(\mathtt{state}, b^{\mathsf{read}})$ are the outputs of the basic CPU-step circuit.
- If $i^{\mathsf{write}} = \perp$ then set $\mathsf{sk}^{\mathsf{write}} := \perp$. Else set $\mathsf{sk}^{\mathsf{write}} := \mathsf{KeyGen}(\mathsf{TSK}_j, \mathsf{id}; r_{\mathsf{write}})$ to be a key for identity $\mathsf{id} = (j, i^{\mathsf{write}}, b^{\mathsf{write}})$.
- Compute $u := \mathsf{WriteTime}(j, \mathtt{state}, i^{\mathsf{read}})$. For $b \in \{0, 1\}$, compute $\mathsf{ct}_b := \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id}_b\ ,\ \mathsf{lbl}_b^{\mathsf{read}};\ r_b)$ where $\mathsf{id}_b = (u, i^{\mathsf{read}}, b)$. Set $\mathsf{translate} := (\mathsf{ct}_0, \mathsf{ct}_1)$.

Figure 1: The Augmented CPU-Step Circuit

circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j)$. The labels for the output wires in each circuit are chosen carefully so that some wire values are revealed in the clear while others remain garbled for the next circuit. See Figure 2 for a useful diagram.

- $(\tilde{P}, k^{\mathsf{in}}) \leftarrow \mathsf{GProg}(P, k, n, t^{init}, t^{cur})$: Let $t^{max} := t^{init} + t^{cur}$. We let j count down from $j = t^{max}$ to $t^{init} + 1$. For each j, we garble $C^P_{\mathsf{CPU}^+}$ by calling $\tilde{C}^P_{\mathsf{CPU}^+}(j) \leftarrow \mathsf{GCircuit}(1^\kappa, C^P_{\mathsf{CPU}^+}, \overline{\mathsf{lbl}})$ where the output labels $\overline{\mathsf{lbl}}$ are chosen as follows:

  - The outputs $i^{\mathsf{read}}, i^{\mathsf{write}}, \mathsf{sk}^{\mathsf{write}}, \mathsf{translate}$ are given out in the clear. For the last circuit $j = t^{max}$, we do not provide these outputs, but instead provide the output $\mathtt{state}'$ in the clear and this serves as the output of the computation. This completely fixes all of the output-wire labels for that circuit.
  - For $j \neq t^{max}$, the labels of the output wires corresponding to $\mathtt{state}'$ are set to match the labels of the input wires corresponding to $\mathtt{state}$ in circuit $\tilde{C}^P_{\mathsf{CPU}^+}(j+1)$.
  - For the initial circuit $j = t^{init} + 1$, we also hard-code the input bit b^{read} to 0.

For $j \neq t^{max}$, let $\mathsf{lbl}_0^{(\mathsf{read},j+1)}$, $\mathsf{lbl}_1^{(\mathsf{read},j+1)}$ be the labels of input wire for the bit b^{read} in the $(j+1)$st garbled circuit. Choose $\mathsf{TSK}_j \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, j)$ and choose fresh encryption randomness $r_0^{(j)}, r_1^{(j)}$ and identity key-generation randomness $r_{\mathsf{write}}^{(j)}$. The jth garbled circuit contains the hard-coded secret values:

$$(\mathsf{MPK}, \mathsf{TSK}_j, j, r_0^{(j)}, r_1^{(j)}, r_{\mathsf{write}}^{(j)}, \mathsf{lbl}_0^{(\mathsf{read},j+1)}, \mathsf{lbl}_1^{(\mathsf{read},j+1)}).$$

Set $\tilde{P} := \left[\tilde{C}^P_{\mathsf{CPU}^+}(t^{init}+1), \ldots, \tilde{C}^P_{\mathsf{CPU}^+}(t^{max})\right]$ to consist of the the t^{cur} garbled circuits created as described above. Finally, set $k^{\mathsf{in}} := \{(i, b, \mathsf{lbl}_b^{\mathsf{in},i}) \ : \ i \in [v], b \in \{0,1\}\}$ to consist of all of the input-wire labels for the v input wires corresponding to the input $\mathtt{state}$ in the initial circuit $\tilde{C}^P_{\mathsf{CPU}^+}(t^{init}+1)$.

Garbled Input. Finally, the garbled input $\tilde{x}$ is created the same way as in garbled circuits. It simply consists of the subset of labels of $k^{\mathsf{in}} := \{(i, b, \mathsf{lbl}_b^{\mathsf{in},i}) \ : \ i \in [v], b \in \{0,1\}\}$ corresponding to the bits of x.

- $\tilde{x} \leftarrow \mathsf{GInput}(x, k^{\mathsf{in}})$: Parse $k^{\mathsf{in}} := \{(i, b, \mathsf{lbl}_b^{\mathsf{in},i}) \ : \ i \in [v], b \in \{0,1\}\}$ and output $\tilde{x} = (\mathsf{lbl}_{x[1]}^{\mathsf{in},1}, \ldots \mathsf{lbl}_{x[v]}^{\mathsf{in},v})$ where $x[i]$ denotes the ith bit of x and $v := |x|$.

Evaluation. To run $y = \mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x})$: parse $\tilde{P} = \left[\tilde{C}^P_{\mathsf{CPU}^+}(1), \ldots, \tilde{C}^P_{\mathsf{CPU}^+}(t)\right]$ as consisting of t garbled circuits. The evaluator evaluates the circuits one-by-one. Set $\tilde{x}_1 := \tilde{x}$. For $j = 1, \ldots, t$:

- When $j = 1$, run $\mathsf{CircEval}(\tilde{C}^P_{\mathsf{CPU}^+}(1), \tilde{x}_1)$ else run $\mathsf{CircEval}(\tilde{C}^P_{\mathsf{CPU}^+}(j), (\tilde{x}_j, \mathsf{lbl}^{\mathsf{read},j}))$ where $\tilde{x}_j$ consists of labels for the garbled $\mathtt{state}_j$, and $\mathsf{lbl}^{\mathsf{read},j}$ is a label for the read-bit in the jth CPU circuit. This reveals the outputs $i_j^{\mathsf{read}}, i_j^{\mathsf{write}}, \mathsf{sk}_j^{\mathsf{write}}, \mathsf{translate}_j = (\mathsf{ct}_0^{(j)}, \mathsf{ct}_1^{(j)})$ in the clear. For $j < t$ it also reveals the garbled output $\tilde{x}_{j+1}$ corresponding to the labels of $\mathtt{state}_{j+1}$ for circuit $j+1$. For $j = t$ it reveals the output of the computation $y = \mathtt{state}_{t+1}$ in the clear.
- Look up $\mathsf{sk}_j^{\mathsf{read}} = \tilde{D}[i_j^{\mathsf{read}}]$, which is a secret key for some identity $\mathsf{id} = (j', i_j^{\mathsf{read}}, b)$, where id can be recovered from the key. Decrypt the ciphertext $\mathsf{ct}_b^{(j)}$ to recover the label $\mathsf{lbl}^{\mathsf{read},j+1} := \mathsf{Dec}_{\mathsf{sk}_j^{\mathsf{read}}}(\mathsf{ct}_b^{(j)})$. Finally, update $\tilde{D}[i_j^{\mathsf{write}}] := \mathsf{sk}_j^{\mathsf{write}}$.

We now state our main technical theorem.

Theorem 5.2. *Given a secure TIBE scheme and a secure circuit garbling scheme with wire labels, the above construction is a UMA (unprotected memory access) secure garbled RAM scheme for all program executions with ptWrites (predictably timed writes).*

Proof Overview. The full proof appears in Appendix C, and here we give a brief overview. The program-simulator simulates $\tilde{P}$ by using the circuit-simulator of the circuit garbling scheme to simulate each of the garbled augmented CPU-step-circuits $\tilde{C}^P_{\mathsf{CPU}^+}(j)$. For the outputs $i_j^{\mathsf{read}}, i_j^{\mathsf{write}}, \mathsf{sk}_j^{\mathsf{write}}$ given by the circuits "in the clear", the simulator provides the correct values using its knowledge of the memory access pattern. However, for the output $\mathsf{translate}$ given by the circuits, it encrypts a "dummy" label for the "wrong" data-bit that the evaluator shouldn't have. We prove indistinguishability using a sequence of hybrids where we change real CPU-step circuits for simulated ones starting from the *first* one. In each hybrid, we can rely on the security of the TIBE ciphertexts given out by $\mathsf{translate}$ even given all of the information needed to create all future garbled circuits. This ensures that there is no "circularity".

Full Security. We give a general transformation from any garbled RAM scheme that only provides UMA security and only supports program executions with ptWrites into a fully secure garbled RAM scheme for arbitrary programs. This transformation uses oblivious RAM (ORAM) to first compile the original program P into a new program P^* that stores/accesses its memory using ORAM. This ensures that the memory contents and access pattern of the compiled program do not reveal anything about those of the original

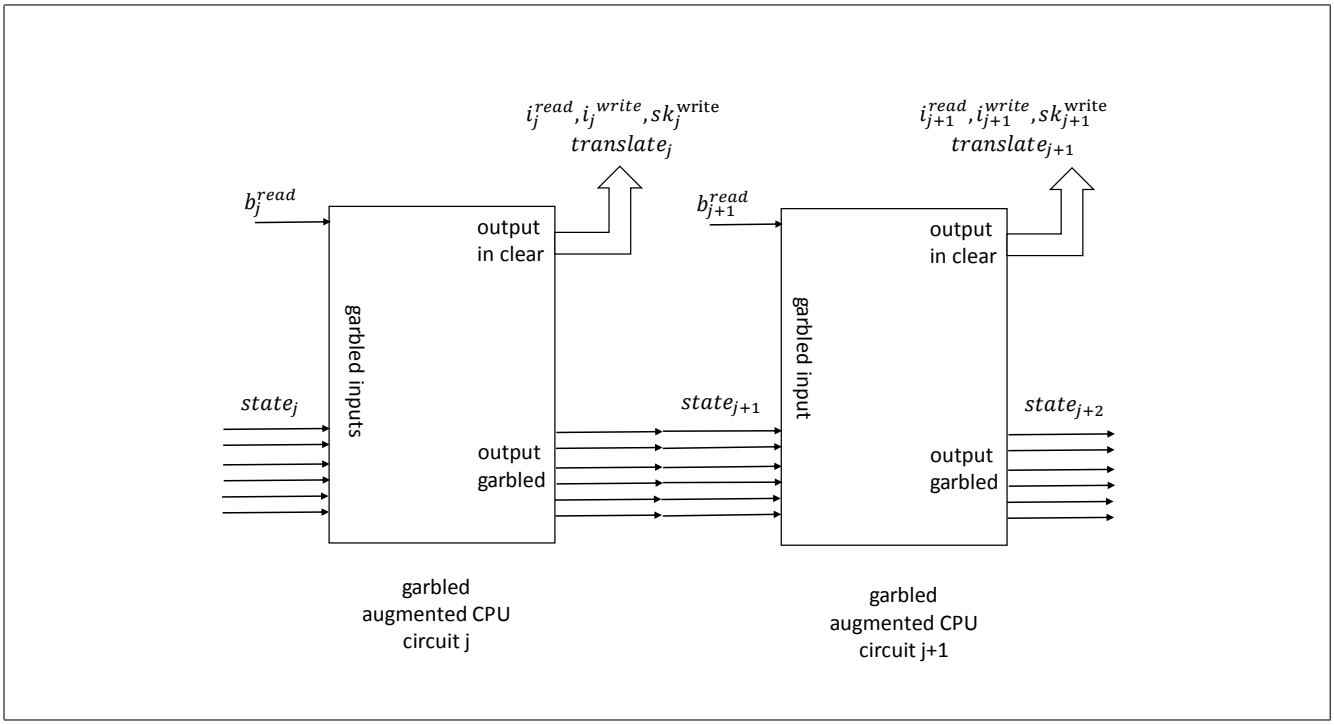

Figure 2: The garbled program consists of t garbled augmented-CPU-circuits. This diagram shows two such circuits and how their inputs/outputs relate to each other. After evaluating garbled circuit j, the evaluator uses the secret key contained in position i_j^{read} of the garbled data to decrypt the translation information $\mathsf{translate}_j$ and recover the correct label for the input wire b_{j+1}^{read} in circuit $j+1$. The evaluator also updates position i_j^{write} of the garbled data with the new key $\mathsf{sk}_j^{\mathsf{write}}$.

program. Some ORAM schemes already ensure that the compiled program satisfies the ptWrites property. If so, we are done. Otherwise, in Appendix A.2 we show how to convert any ORAM scheme into one that satisfies the ptWrites property at an additional $O(\log n)$ overhead. Now we can simply apply our original UMA-secure garbled RAM scheme for ptWrites on this compiled program to get a fully secure solution. This gives us our main result, whose proof appears in Appendix D.

Theorem 5.3. *Assuming the existence of IBE, there exists a garbled RAM scheme.*

6 Applications and Optimizations

We briefly mention several extensions, optimizations and applications of our garbled RAM. Many of these were also discussed by [LO13] in the context of their scheme.

Most importantly, the garbled RAM scheme supports *wire labeling*, where we can assign arbitrary labels to the output wires of the garbled program, the input-garbling key k^{in} consists of a collection of labels for the input wires, and the garbled input $\tilde{x}$ consists of the subset of the labels in k^{in} corresponding to the input bits. This gives us the two applications in the next two paragraphs.

Private/Verifiable Output. We can make the output of the program private by choosing random labels for the output wires of the last CPU circuit, rather than providing the output in the clear. The evaluator has to send these labels back to communicate the output without learning it. We can also get verifiability in this way: if the evaluator is able to produce some output labels, we know they must correspond to the correct output of the computation.

Two-Party Computation. The garbled input $\tilde{x}$ can be exchanged using oblivious transfer (OT) in the context of two party computation. In particular, party 1 sends OT queries corresponding to its input x and party 2 sends the garbled data/program $\tilde{D}, \tilde{P}$ and the OT responses which reveal exactly $\tilde{x}$. This gives a us secure 2-party computation of a RAM program with minimal interaction. Multiparty computation of RAM programs was previously studied by [OS97, GKK+12].

Efficiency Optimization. In our definition, we required that the size/evaluation time of the garbled program is $|C^P_{\mathsf{CPU}}| \cdot t \cdot \mathsf{poly}(\kappa) \cdot \mathsf{polylog}(n)$ where t is the original running time. The CPU-step-circuit C^P_{CPU} may have a large description if the program P, the input x or the output y are large: $|C^P_{\mathsf{CPU}}| > (|P|+|x|+|y|)$. In this case, rather than just garbling the program P directly, it is better to garble/execute a sequence of programs that (1) Write P and x to memory (each program can write 1 bit), (2) execute a single "universal RAM" program that runs the code P contained in memory and writes the output y to memory, and (3) read y from memory (each program outputs 1 bit). This gives a total complexity $(|P| + |x| + |y| + t)\mathsf{poly}(\kappa)\mathsf{polylog}(n)$ rather than $(|P| + |x| + |y|)t\mathsf{poly}(\kappa)\mathsf{polylog}(n)$.

7 Conclusions

We conclude with two important open problems. Firstly, it would be interesting to give a garbled RAM scheme with polylogarithmic overhead based only on the existence of one-way functions. Secondly, the work of Goldwasser et al. recently constructed the first *reusable* garbling schemes for *circuits* and *Turing machines* [GKP+13b, GKP+13a] 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.

References

[BB11] Dan Boneh and Xavier Boyen. Efficient selective identity-based encryption without random oracles. *J. Cryptology*, 24(4):659–693, 2011.

[BDWY12] Mihir Bellare, Rafael Dowsley, Brent Waters, and Scott Yilek. Standard security does not imply security against selective-opening. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 645–662. Springer, 2012.

[BF03] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003.

[BHR12a] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT*, pages 134–153, 2012.

[BHR12b] 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.

[BHY09] Mihir Bellare, Dennis Hofheinz, and Scott Yilek. Possibility and impossibility results for encryption and commitment secure under selective opening. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2009.

[BRS02] John Black, Phillip Rogaway, and Thomas Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In Kaisa Nyberg and Howard M. Heys, editors, *Selected*

Areas in Cryptography, volume 2595 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2002.

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

[GGH+13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In Emiliano De Cristofaro and Matthew Wright, editors, *Privacy Enhancing Technologies*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[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+13a] 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 (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 536–553. Springer, 2013.

[GKP+13b] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *STOC*, pages 555–564. ACM, 2013.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP (2)*, volume 6756 of *Lecture Notes in Computer Science*, pages 576–587. Springer, 2011.

[GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In Yuval Rabani, editor, *SODA*, pages 157–167. SIAM, 2012.

[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 V. Aho, editor, *STOC*, pages 182–194. ACM, 1987.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble ram programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer, 2013.

[LO14] Steve Lu and Rafail Ostrovsky. Garbled ram revisited, part II. ePrint archive report, 2014.

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

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage. In *STOC*, 1997.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious rams. In Harriet Ortiz, editor, *STOC*, pages 514–523. ACM, 1990.

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

[PR10] Benny Pinkas and Tzachy Reinman. Oblivious RAM Revisited. In *CRYPTO*, pages 502–519, 2010.

[Rot13] Ron Rothblum. On the circular security of bit-encryption. In *TCC*, pages 579–598, 2013.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[SS13] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious distributed cloud data store. In *NDSS*. The Internet Society, 2013.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious ram. In *NDSS*. The Internet Society, 2012.

[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. *IACR Cryptology ePrint Archive*, 2013:280, 2013. To appear in ACM CCS 2013.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.

A Building Blocks: Garbled Circuit & Oblivious RAM

A.1 Garbled Circuits with Wire-Labels

As a tool, we rely on standard Yao garbled circuits introduced by [Yao82] and recently formalized/abstracted by [LP09, BHR12b, BHR12a]. We abstract out the properties of garbled circuits that we need via the notion of a garbled circuit with *wire labels*. In such a scheme, each input and output wire w of the circuit is associated with two *labels* $\mathsf{lbl}_0^w, \mathsf{lbl}_1^w$ corresponding to the bit-values $0, 1$. The garbling scheme can be given an arbitrary assignment of *output wires* labels (this need not be random; for example it is legitimate to set $\mathsf{lbl}_0^w = 0, \mathsf{lbl}_1^w = 1$ so as to reveal the output value on wire w in the clear). The garbling scheme outputs a garbled circuit $\tilde{C}$ and an assignment of labels for the input wires. Given a garbled circuit $\tilde{C}$ and input-wire labels corresponding to some input x, the evaluator can learn the output-wire labels corresponding to the output $y = C(x)$, but nothing else.

More formally, a circuit garbling scheme with wire labels consists of two algorithms:

- $(\tilde{C},\ \{\ (j, b, \mathsf{lbl}_b^{\mathsf{in},j})\ \}\ ) \leftarrow \mathsf{GCircuit}(1^\kappa, C,\ \{\ (i, b, \mathsf{lbl}_b^{\mathsf{out},i})\ \}\ )$: Given a circuit C with input size v_{in} and output size v_{out}, and a set of *output labels* $\mathsf{lbl}_b^{\mathsf{out},i}$ for all output wires $i \in [v_{\mathsf{out}}]$ and $b \in \{0, 1\}$, outputs a *garbled circuit* $\tilde{C}$ and a set of *input labels* $\mathsf{lbl}_b^{\mathsf{in},j}$ for every input wire $j \in [v_{\mathsf{in}}]$ and $b \in \{0, 1\}$.
- $(\mathsf{lbl}^{\mathsf{out},1}, \ldots, \mathsf{lbl}^{\mathsf{out},v_{\mathsf{out}}}) = \mathsf{Eval}(\tilde{C}, (\mathsf{lbl}^{\mathsf{in},1}, \ldots, \mathsf{lbl}^{\mathsf{in},v_{\mathsf{in}}}))$: Given a garbled circuit $\tilde{C}$ and a sequence of input labels $\mathsf{lbl}^{\mathsf{in},j}$, outputs a sequence of output labels $\mathsf{lbl}^{\mathsf{out},i}$. Intuitively, if the input labels correspond to some input $x \in \{0, 1\}^{v_{\mathsf{in}}}$ then the output labels should correspond to $y = C(x)$.

Correctness. For correctness, we require that for any circuit C and any input $x \in \{0, 1\}^{v_{\mathsf{in}}}, x = (x[1], \ldots, x[v_{\mathsf{in}}])$ such that $y = (y[1], \ldots, y[v_{\mathsf{out}}]) = C(x)$ and any set of output labels $\{\ (i, b, \mathsf{lbl}_b^{\mathsf{out},i})\ \}$ we have

$$\Pr\left[\ \mathsf{Eval}(\tilde{C}, (\mathsf{lbl}_{x[1]}^{\mathsf{in},1}, \ldots, \mathsf{lbl}_{x[v_{\mathsf{in}}]}^{\mathsf{in},v_{\mathsf{in}}}))\ =\ (\mathsf{lbl}_{y[1]}^{\mathsf{out},1}, \ldots, \mathsf{lbl}_{y[v_{\mathsf{out}}]}^{\mathsf{out},v_{\mathsf{out}}})\ \right] = 1.$$

where $(\tilde{C},\ \{\ (j, b, \mathsf{lbl}_b^{\mathsf{in},j})\ \}\ ) \leftarrow \mathsf{GCircuit}(C,\ \{\ (i, b, \mathsf{lbl}_b^{\mathsf{out},i})\ \}\ )$.

Security. For security, we require that there is a PPT simulator Sim such that for any $C, x, \{\ (i, b, \mathsf{lbl}_b^{\mathsf{out},i})\ \}$ as above, we have

$$(\ \tilde{C}\ ,\ \mathsf{lbl}_{x[1]}^{\mathsf{in},1}, \ldots, \mathsf{lbl}_{x[v_{\mathsf{in}}]}^{\mathsf{in},v_{\mathsf{in}}}\ ) \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\kappa, C\ ,\ \mathsf{lbl}_{y[1]}^{\mathsf{out},1}, \ldots, \mathsf{lbl}_{y[v_{\mathsf{out}}]}^{\mathsf{out},v_{\mathsf{out}}}\ )$$

where $(\tilde{C},\ \{\ (j, b, \mathsf{lbl}_b^{\mathsf{in},j})\ \}\ ) \leftarrow \mathsf{GCircuit}(C,\ \{\ (i, b, \mathsf{lbl}_b^{\mathsf{out},i})\ \}\ )$, $y = C(x)$.

Notes & Conventions. We say that an output wire w is given ***in the clear*** if its labels of are set to $\mathsf{lbl}_0^w = 0, \mathsf{lbl}_1^w = 1$. Sometimes, it will be useful to think of the garbled circuit as containing some ***hard-coded*** secret value (for example a cryptographic key). Notice that, in our definition, the circuit itself is considered to be public and so this may seem problematic. However, we can always include a hard-coded secret value by thinking of this value as part of the input to the circuit, but including the appropriate labels for the desired value of this input together with the garbled circuit. Lastly, notice that our definition makes it easy to ***sequentially compose*** several garbled circuits where some of the outputs of one circuit are fed as inputs to the next circuit. This is done by setting the output labels of the first circuit to match the input labels of the next circuit.

A.2 Oblivious RAM

Oblivious RAM [Gol87, Ost90, GO96] allows a user to encode some data D into a special format D^* in such a way that (1) we can emulate an access (read/write) to D by accessing poly-logarithmically many locations of D^* and (2) the resulting access pattern in D^* does not reveal anything about the intended access pattern of D. We will also insist that D^* hides the contents of D, which can always be done by encrypting the contents. Oblivious RAM has received much attention recently in the works of [PR10, GM11, SCSL11, SSS12, GMOT12, SS13, GGH$^+$13, SvDS$^+$13, LO13]. We use a slightly non-standard notation for oblivious RAM to match our notation of RAM computation in Section 2, but the notion is equivalent to standard definitions.

Definition. An oblivious RAM scheme consists of procedures $(\mathsf{OData}, \mathsf{OProg})$ with syntax:

- $D^* \leftarrow \mathsf{OData}(D, k)$: Takes data D and secret key k and outputs encoded data D^*.
- $P^* \leftarrow \mathsf{OProg}(P)$: Outputs a compiled program P^* which works over the encoded data. The compiled program now expects an additional input k to access this data and computes the same function as the original program: $(P^*)^{D^*}(x, k) = P^D(x)$

For any data D, programs $P_1, \ldots, P_\ell$ with run-times $t_1, \ldots, t_\ell$ and all inputs $x_1, \ldots, x_\ell$ consider the experiment of selecting $k \leftarrow \{0,1\}^\kappa, D^* \leftarrow \mathsf{OData}(D, k), \{P_i^* := \mathsf{OProg}(P_i)\}$. Then we require:

- **Correctness:** $\Pr[(P_1^*(x_1, k), \ldots, P_\ell^*(x_\ell, k))^{D^*} = (P_1(x_1), \ldots, P_\ell(x_\ell))^D] = 1$.
- **Security:** There exists a universal simulator Sim such that $\mathsf{Sim}(1^\kappa, n, \{t_i\}_{i=1}^\ell) \overset{\text{comp}}{\approx} (D^*, \mathsf{MemAccess})$. where $\mathsf{MemAccess} = \{(i_j^{\mathsf{read}}, i_j^{\mathsf{write}}, b_j^{\mathsf{write}})\}$ consists of all of the memory access outputs of the CPU-step circuits during the execution of the compiled computation $(P_1^*(x_1, k), \ldots, P_\ell^*(x_\ell, k))^{D^*}$.
- **Efficiency:** the running time t_i^* of each program $P_i^*(x_i, k)$ in the context of the execution $(P_1^*(x_1, k), \ldots, P_\ell^*(x_\ell, k))^{D^*}$ is at most $t_i^* = t_i \mathsf{poly}(\kappa)\mathsf{polylog}(n)$. Furthermore, one can efficiently compute t_i^* from t_i.

Notes. The above requirements implicitly assume oblivious RAM with *worst-case* poly-logarithmic overhead since we require the efficiency of *each* program to only be poly-logarithmically larger – this includes the case where the program only performs one memory access operation. We could also define *amortized* efficiency, but fortunately ORAM schemes with worst-case efficiency are known (e.g., [OS97, SCSL11]). The above also assumes that the ORAM does not keep any "long-term" state since each fresh program execution only gets the secret key k. This is always possible to achieve by encrypting any necessary state, storing it on the server at the end of any program execution, and reading it back from the server at the beginning of any program execution. Notice that the above (implicitly) only considers a passive attacker that passively observes the memory access pattern during an honest computation but does not provide any incorrect values. Perhaps surprisingly, this is sufficient for our needs even though, in the context of garbled RAM, the attacker need not run the garbled computation honestly. However, the use of garbled circuits implicitly already ensures that active attacks are not possible.

A.3 Oblivious RAM with Predictably Timed Writes

We now show how to take any oblivious RAM scheme and also ensure that the compiled program execution satisfies the predictably timed writes (ptWrites) property (Definition 4.1). Some ORAM schemes from the literature already satisfy ptWrites, as is the case with the hierarchical scheme of Goldreich and Ostrovsky [GO96].[6] However, we show that ptWrites can also be easily and generically added to any ORAM scheme at an additional $O(\log n)$ cost. Actually, our transformation isn't specific to oblivious RAM at all – it shows how to compile any program into one that satisfies ptWrites. Since it only looks at the memory access pattern of the original program and not at its internals, this compilation does not "break" obliviousness.

Description. The main idea of our compiler is simple: we add a binary tree on top of the original data where the leafs of the tree correspond to the bits of the original data and each internal node of the tree contains the "last-write-time" of each of its two children (initially, these are all set to 0). Whenever we want to read a value in the original data, we follow the corresponding path down the tree. The last-write time of the root corresponds to the total number of write operations performed so far, which we can remember in the state of the computation. Whenever we read a node, we temporarily remember the last-write-time of its children (by keeping this info in the state). This ensures that *before* we read the contents of any node in the tree, including the actual data at the leafs, we know its last-write time. To write to some location in the original data, we follow the same procedure as in the case of a read, but after we read the values in each node, we also increment the last-write-time for the corresponding child on the path to the leaf.

In the context of oblivious RAM, we *first* apply the ORAM encoding to convert the original data D into the ORAM encoded data D^* and *then* apply the above transformation to access values inside D^*. This makes sure that we do not harm the "oblivious" nature of the computation (we will not reveal anything more than the access inside D^*) while maintaining the ptWrites property of the final scheme.

B Constructing TIBE from Standard IBE

We now show how to construct a timed IBE (TIBE) scheme (Definition 5.1) from any selectively-secure IBE scheme. See [BF03, BB11] for a standard definition of IBE and selectively-secure IBE. In fact, we can even rely on a weak form of selective security where the challenge identity *as well as* all of the key-query identities for which the attacker requests to see secret keys are chosen non-adaptively before seeing MPK.

High-Level Description. For simplicity, we assume (w.l.o.g.) that the identities of the TIBE scheme are of the form (j, v) where $v \in \{0,1\}^\kappa$ since we can always apply collision-resistant hashing to reduce the size of v to just κ bits. The high level idea is to make the time-period keys TSK_j for $j > 0$ consists of 2κ identity secret keys of the IBE scheme: $\mathsf{TSK}_j = \{\mathsf{sk}_{(j,\alpha,b)} \ : \ \alpha \in [\kappa], b \in \{0,1\}\}$. The secret key for identity (j, v) in the TIBE scheme will consist of a subset of TSK_j depending on the bits of v: more precisely, we set $\mathsf{sk}'_{(j,v)} = (\mathsf{sk}_{(j,1,v[1])}, \ldots, \mathsf{sk}_{(j,\kappa,v[\kappa])})$. The above crucially relies on the fact that we give out only one secret key for each time period $j > 0$ to ensure that for any identity (j, v') with $v' \neq v$ the adversary will be missing at least one of the value $\mathsf{sk}_{(j,\alpha,v'[\alpha])}$. To encrypt to an identity (j, v) the TIBE scheme will (κ-out-of-κ)-secret share the message and encrypt each share $\alpha \in [\kappa]$ under the IBE identity $(j, \alpha, v[\alpha])$ to ensure that the decryptor must have all of these keys. Finally, time period $j = 0$ is treated as special: we set $\mathsf{TSK}_0 := \mathsf{MSK}$ to be the master key of the IBE, and the TIBE identity-secret-keys for identities $(0, v)$ are simply equal to the IBE identity-secret-keys for the same identity.

Detailed Description. In more detail, let $\Pi = (\mathsf{MasterGen}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be any IBE scheme. We define a TIBE scheme $\Gamma = (\mathsf{MasterGen}, \mathsf{TimeGen}, \mathsf{KeyGen}', \mathsf{Enc}', \mathsf{Dec}')$ as follows:

- The $\mathsf{MasterGen}$ procedure of the TIBE scheme is the same as that of the IBE scheme.

[6] Indeed, the work analyzed a similar but slightly different property called "time-labeled simulation".

- $\mathsf{TSK}_j \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, j)$: If $j = 0$, set $\mathsf{TSK}_0 := \mathsf{MSK}$. Else set

$$\mathsf{TSK}_j := \{\mathsf{sk}_{(j,\alpha,b)} \ : \ \alpha \in [\kappa], b \in \{0,1\}\}$$

  where, for each $\mathsf{id} = (j, \alpha, b)$, we choose $\mathsf{sk}_{\mathsf{id}} \leftarrow \mathsf{KeyGen}(\mathsf{MSK}, \mathsf{id})$.

- $\mathsf{sk}'_{(j,v)} \leftarrow \mathsf{KeyGen}'(\mathsf{TSK}_j, (j, v))$: If $j = 0$, parse $\mathsf{TSK}_0 = \mathsf{MSK}$ and output $\mathsf{sk}'_{(0,v)} \leftarrow \mathsf{KeyGen}(\mathsf{MSK}, (0, v))$. Else parse $\mathsf{TSK}_j := \{\mathsf{sk}_{(j,\alpha,b)}\}$ and set $\mathsf{sk}'_{(j,v)} := (\mathsf{sk}_{(j,1,v[1])}, \ldots, \mathsf{sk}_{(j,\kappa,v[\kappa])})$.

- $\mathsf{ct}' \leftarrow \mathsf{Enc}'_{\mathsf{MPK}}((j, v), \mathsf{msg})$: If $j = 0$, simply output $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((0, v), \mathsf{msg})$. Otherwise, chooses a ($\kappa$-out-of-$\kappa$)-additive-secret-sharing $\mathsf{share}_1, \ldots, \mathsf{share}_\kappa$ of the message msg and, for $\alpha \in [\kappa]$ compute $\mathsf{ct}_\alpha \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((j, \alpha, v[\alpha]), \mathsf{share}_\alpha)$. Output $\mathsf{ct}' = (\mathsf{ct}_1, \ldots, \mathsf{ct}_\kappa)$.

- $\mathsf{msg} = \mathsf{Dec}'_{\mathsf{sk}'_{(j,v)}}(\mathsf{ct}')$: If $j = 0$, simply output $\mathsf{msg} = \mathsf{Dec}_{\mathsf{sk}'_{(0,v)}}(\mathsf{ct}')$. Else parse $\mathsf{ct}' = (\mathsf{ct}_1, \ldots, \mathsf{ct}_\kappa)$, $\mathsf{sk}'_{(j,v)} = (\mathsf{sk}_{(j,1,v[1])}, \ldots, \mathsf{sk}_{(j,\kappa,v[\kappa])})$, compute $\mathsf{share}_\alpha := \mathsf{Dec}_{\mathsf{sk}_{(j,\alpha,v[\alpha])}}(\mathsf{ct}_\alpha)$, and output the recovered value msg by combining the shares $(\mathsf{share}_1, \ldots, \mathsf{share}_\kappa)$.

Theorem B.1. *Assume that* Π *is a selectively-secure IBE scheme. Then the scheme* Γ *described above is a secure TIBE.*

Proof. Let $\mathcal{A}$ be an adversary in the TIBE security game. We construct an attacker $\mathcal{B}$ on the selective-security of the IBE scheme. Assume that, in the first round, $\mathcal{A}$ chooses the challenge TIBE identity $\mathsf{id}^* = (j^*, v^*)$ and some set of TIBE identities $S = S_0 \cup S_{>0}$ satisfying the needed condition, some value $t \geq j^*$ and some message $\mathsf{msg}_0, \mathsf{msg}_1$. The attacker $\mathcal{B}$ simply asks for all of the IBE identity-secret-keys needed to (honestly) generate TIBE identity-secret-keys in the set S as well as the values $\mathsf{TSK}_{j^*+1}, \ldots, \mathsf{TSK}_t$ which it will give to $\mathcal{A}$. Let's call the set of identities that $\mathcal{B}$ queries S'. We will do a case analysis depending on whether $j^* = 0$.

- If $j^* = 0$, then $\mathcal{B}$ simply chooses the challenge identity $\mathsf{id}^* = (j^*, v^*)$ and the messages $\mathsf{msg}_0, \mathsf{msg}_1$ to give to its challenger. It is easy to see that $\mathsf{id}^* \notin S'$. The attacker $\mathcal{B}$ get back a ciphertext ct, gives it to $\mathcal{A}$ and outputs the bit $\hat{b}$ output by $\mathcal{A}$. It is easy to see that the probability of $\mathcal{B}$ winning the IBE selective-security game is exactly the same as that of $\mathcal{A}$ winning the TIBE security game.

- If $j^* \neq 0$, the set S might contain exactly one identity of the form (j^*, v) where $v \neq v^*$. Let α^* be some bit such that $v[\alpha^*] \neq v^*[\alpha^*]$. The attacker $\mathcal{B}$ chooses the challenge identity $\tilde{\mathsf{id}}^* = (j^*, \alpha^*, v^*[\alpha^*])$. It is easy to see that $\tilde{\mathsf{id}}^* \notin S'$. The attacker $\mathcal{B}$ chooses random shares share_α for all $\alpha \neq \alpha^*$ and computes $\mathsf{share}^0_{\alpha^*}$ to complete the sharing of msg_0 and $\mathsf{share}^1_{\alpha^*}$ to complete the sharing of msg_1. The attacker $\mathcal{B}$ also chooses ciphertexts $\mathsf{ct}_\alpha \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((j^*, \alpha, v^*[\alpha]), \mathsf{share}_\alpha)$ for all $\alpha \neq \alpha^*$. It gives the messages $\mathsf{share}^0_{\alpha^*}, \mathsf{share}^1_{\alpha^*}$ to its challenger and gets back a ciphertext ct_{α^*}. It gives $\mathcal{A}$ the ciphertexts $(\mathsf{ct}_1, \ldots, \mathsf{ct}_\kappa)$ and output the bit $\hat{b}$ output by $\mathcal{A}$. It is easy to see that the probability of $\mathcal{B}$ winning the IBE selective-security game is exactly the same as that of $\mathcal{A}$ winning the TIBE security game.

□

C Proof of Theorem 5.2

Correctness and efficiency follow directly from the definition. Therefore we focus on security. Let $\mathsf{CircSim}$ be the simulator of the circuit garbling scheme. We define a simulator Sim for our garbled program as follows:

Input: Sim gets the inputs $\{P_a, t_a, y_a\}_{a=1}^{\ell}, D, \mathsf{MemAccess} = \{(i_j^{\mathsf{read}}, i_j^{\mathsf{write}}, b_j^{\mathsf{write}})\}_{j=1}^{t^{max}}$, where program P_a executes t_a CPU steps and output y_a, the initial memory contents are $D \in \{0,1\}^n$ and $\mathsf{MemAccess}$ describes the entire memory access throughout all $t^{max} = \sum_{a=1}^{\ell} t_a$ CPU steps executed.

Output: The simulator Sim outputs: $\tilde{D}, \tilde{P}_1, \ldots, \tilde{P}_\ell, \tilde{x}_1, \ldots, \tilde{x}_\ell$.

Initialization: The simulator Sim uses the TIBE scheme to choose $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MasterGen}(1^\kappa)$ and $\mathsf{TSK}_j \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, j)$ for $j \in \{0, \ldots, t^{max}\}$.

Garbled Data: It creates $\tilde{D}$ using the honest process, by choosing $\tilde{D}[i] \leftarrow \mathsf{KeyGen}(\mathsf{TSK}_0, (0, i, D[i]))$ to be a secret key for the identity $(0, i, D[i])$.

Garbled Programs/Inputs: The simulator Sim processes each program $a \in [\ell]$ separately. Let $j_a^{init} = \sum_{b=1}^{a-1} t_a + 1$ be the first CPU step executed by the program P_a and let $j_a^{max} = j_a^{init} + t_a - 1$ be the last step. The simulator starts from $j = j_a^{max}$ and counts down to $j = j_a^{init}$. For each j:

- For the last circuit $j = j_a^{max}$, create $\tilde{C}^P_{\mathsf{CPU}^+}(j)$ by calling $\mathsf{CircSim}$ on the circuit $C^P_{\mathsf{CPU}^+}$ with the output-labels of $\mathtt{state}_{j+1}$ set to the value y_a in the clear. This produces some set of input labels for the input $\mathtt{state}_j$ and the bit b_j^{read}.
- For any other $j \neq j_{max}$, create $\tilde{C}^P_{\mathsf{CPU}^+}(j)$ by calling $\mathsf{CircSim}$ on the circuit $C^P_{\mathsf{CPU}^+}$ where the output-labels for $i_j^{\mathsf{read}}, i_j^{\mathsf{write}}, \mathsf{sk}_j^{\mathsf{write}}, \mathsf{translate}$ are given "in the clear" and the output-labels of of the updated $\mathtt{state}_{j+1}$ are set to match the input labels for $\mathtt{state}_{j+1}$ given by the circuit-simulator for the circuit $j+1$. The actual values $\mathsf{sk}_j^{\mathsf{write}}, \mathsf{translate}$ are computed via:
  - If $i_j^{\mathsf{write}} = \perp$ then set $\mathsf{sk}_j^{\mathsf{write}} := \perp$. Else choose $\mathsf{sk}_j^{\mathsf{write}} \leftarrow \mathsf{KeyGen}(\mathsf{TSK}_j, \mathsf{id} = (j, i_j^{\mathsf{write}}, b_j^{\mathsf{write}}))$.
  - Let $u < j$ be the last write-time to location i_j^{read} (i.e., the largest value such that $i_u^{\mathsf{write}} = i_j^{\mathsf{read}}$) and let $b = b_u^{\mathsf{write}}$ be the bit written to the location at time u (this can be easily computed given $\mathsf{MemAccess}$). Set:

    $$\mathsf{ct}_b \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((u, i_j^{\mathsf{read}}, b)\ ,\ \mathsf{lbl}^{\mathsf{read},j+1}) \quad , \quad \mathsf{ct}_{1-b} \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((u, i_j^{\mathsf{read}}, 1-b)\ ,\ 0^{v_{\mathsf{lbl}}})$$

    where $\mathsf{lbl}^{\mathsf{read},j+1}$ is the label of the "read-bit" wire given by the circuit-simulator for the circuit $j+1$ and v_{lbl} is the label-length. Set $\mathsf{translate} := (\mathsf{ct}_0, \mathsf{ct}_1)$.

Set $\tilde{x}$ to be the input labels created by the circuit-simulator for the input $\mathtt{state}$ of the initial circuit.

We now need to argue the indistinguishability of the real output and the simulation. To do so, we define a series of hybrid distributions $\mathbf{Hyb}_j$ for $j = 1, \ldots, t^{max}$. In the hybrid j, garbled circuits $1, \ldots, j$ are created as in the simulation and garbled circuits $j+1, \ldots, t^{max}$ are created as in the real distribution. We do not distinguish between which CPU steps belong to which program and are oblivious to program boundaries, except that in each hybrid where we switch the initial circuit of some program from real to simulated, we also switch the garbled input $\tilde{x}$ for that program to be simulated as well. In $\mathbf{Hyb}_j$, when we simulate the jth circuit, we use the output labels for $\mathtt{state}_{j+1}$ and $\mathsf{lbl}^{\mathsf{read},j+1}$ to match the labeling in garbled circuit $j+1$, for the *actual* value of $\mathtt{state}_{j+1}, b_{j+1}^{\mathsf{read}}$ that these wires take on during the real computation.

We also define a hybrid distribution $\mathbf{Hyb}'_j$ which is like $\mathbf{Hyb}_j$ except for the simulation of the jth CPU-step circuit. Instead of choosing $\mathsf{translate}$ as in the simulation described above, we choose $\mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1)$ to both be encryptions of the correct label of the next circuit:

$$\mathsf{ct}_0 \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((u, i_j^{\mathsf{read}}, 0)\ ,\ \mathsf{lbl}_0^{\mathsf{read},j+1}) \quad , \quad \mathsf{ct}_1 \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((u, i_j^{\mathsf{read}}, 1)\ ,\ \mathsf{lbl}_1^{\mathsf{read},j+1})$$

where $\mathsf{lbl}_0^{\mathsf{read},j+1}, \mathsf{lbl}_1^{\mathsf{read},j+1}$ are the labels corresponding to the bits 0,1 for the wire b^{read} in circuit $j+1$ which is still created using the real garbling procedure. (When the CPU step j is the end of a program execution and does not output $\mathsf{translate}$, we just define $\mathbf{Hyb}'_j$ to be the same as $\mathbf{Hyb}_j$.)

Notice that $\mathbf{Hyb}_0$ is equal to the real distribution and $\mathbf{Hyb}_{t^{max}}$ is equal to the simulated distribution. Therefore, we prove the theorem by showing that for each j, we have:

$$\mathbf{Hyb}_j \overset{\text{comp}}{\approx} \mathbf{Hyb}'_{j+1} \overset{\text{comp}}{\approx} \mathbf{Hyb}_{j+1}$$

We prove this in the following two claims.

Claim C.1. *For each $j \in \{0, \ldots, t^{max}\}$ we have* $\mathbf{Hyb}_j \overset{\text{comp}}{\approx} \mathbf{Hyb}'_{j+1}$.

Proof. This follows directly from the security of the circuit-garbling scheme applied only to the garbled augmented-CPU circuit $j+1$. This is because, in $\mathbf{Hyb}_j$, all of the circuits $1, \ldots, j$ are already simulated and hence they only rely on a subset of the input-wire labels for the inputs $\mathtt{state}_{j+1}, b^{\mathsf{read}}_{j+1}$ in circuit $j+1$ corresponding to the actual values that these wires should take on during the real computation. (This is true for the wire corresponding to b^{read}_{j+1} since the simulated translation map $\mathsf{translate}$ used to create the jth circuit only encrypts one label and the other ciphertext is "dummy". Indeed, the above holds even given all of the secrets of the TIBE scheme.) □

Claim C.2. *For each $j \in \{1, \ldots, t^{max}\}$ we have* $\mathbf{Hyb}'_j \overset{\text{comp}}{\approx} \mathbf{Hyb}_j$.

Proof. This follows directly from the security of the TIBE scheme. The only difference between $\mathbf{Hyb}'_j$ and $\mathbf{Hyb}_j$ is the value of $\mathsf{translate}_j = (\mathsf{ct}_0, \mathsf{ct}_1)$ used to simulate the jth garbled circuit. Let $b = b^{\mathsf{read}}_{j+1}$ be the *value* of the read-bit in location i^{read}_j in the real computation. Then, in $\mathbf{Hyb}'_j$ we set

$$\mathsf{ct}_b \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((u, i^{\mathsf{read}}_j, b)\ ,\ \mathsf{lbl}^{\mathsf{read},j+1}_b) \quad , \quad \mathsf{ct}'_{1-b} \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((u, i^{\mathsf{read}}_j, 1-b)\ ,\ \mathsf{lbl}^{\mathsf{read},j+1}_{1-b})$$

whereas in $\mathbf{Hyb}_j$ we set

$$\mathsf{ct}_b \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((u, i^{\mathsf{read}}_j, b)\ ,\ \mathsf{lbl}^{\mathsf{read},j+1}_b) \quad , \quad \mathsf{ct}_{1-b} \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((u, i^{\mathsf{read}}_j, 1-b)\ ,\ 0^{v_{\mathsf{lbl}}})$$

where $u < j$. Therefore we need to argue that semantic security holds for the ciphertext ct_{1-b} encrypted for identity $\mathsf{id}^* == (u, i^{\mathsf{read}}_j, 1-b)/$

Notice that the only values related to the TIBE scheme that are available in $\mathbf{Hyb}_j, \mathbf{Hyb}'_j$ are:

- The master public key MPK which is hard-coded into all the garbled circuits.
- Time-period secret keys $\mathsf{TSK}_{j+1}, \ldots, \mathsf{TSK}_{t^{max}}$ which are hard-coded into all of the future "real" garbled circuits.
- For each time period $j' : 0 < j' \leq j$, a single identity secret key for the identity $(j', i_{j'}, b_{j'})$ where the bit $b_{j'}$ was written to location $i_{j'}$ in step j' of the actual computation. This key is sued to simulate circuit j'.
- For time period $j' = 0$, the various keys that were used to create $\tilde{D}$.

The identities of the keys given above differ from the identity $\mathsf{id}^* = (u, i^{\mathsf{read}}_j, 1-b)$. Moreover, the above values satisfy the requirements of the TIBE definition (Definition 5.1). Therefore we can rely on the security of the TIBE scheme to argue that ct_{1-b} and ct'_{1-b} are indistinguishable. □

This concludes the proof of the theorem.

D Upgrading UMA/ptWrites to Full Security/Functionality

We now describe a general transformation from any garbled RAM scheme that only provides UMA security and only supports program executions with ptWrites into a fully secure garbled RAM scheme for arbitrary programs. This transformation uses oblivious RAM (ORAM) to first compile the original program P into a new program P^* that stores/accesses its memory using ORAM. This ensures that the memory contents and access pattern of the compiled program do not reveal anything about those of the original program. Some ORAM schemes already ensure that the compiled program satisfies the ptWrites property. If so, we are done. Otherwise, in Appendix A.2 we show how to convert any ORAM scheme into one that satisfies the ptWrites property at an additional $O(\log n)$ overhead. Now we can simply apply our original UMA-secure garbled RAM scheme for ptWrites on this compiled program to get a fully secure solution.

The Compiler. Let $G = (\mathsf{GData}, \mathsf{GProg}, \mathsf{GInput}, \mathsf{GEval})$ be any garbled RAM scheme that provides UMA security and only supports program executions with ptWrites. Let $O = (\mathsf{OData}, \mathsf{OProg})$ be any ORAM that guarantees ptWrites. We define a garbled RAM scheme G' which first applies the ORAM O to the program to make it oblivious and satisfy ptWrites, and then uses G to garbled it. In detail, we define $G' = (\mathsf{GData}', \mathsf{GProg}', \mathsf{GInput}', \mathsf{GEval})$ as follows:

- $\mathsf{GData}'(D, k = (k_1, k_2))$: Call $D^* \leftarrow \mathsf{OData}(D, k_1), \tilde{D}^* \leftarrow \mathsf{GData}(D^*, k_2)$. Output $\tilde{D}^*$.
- $\mathsf{GProg}'(P, k = (k_1, k_2)), n, t_{init}, t_{cur})$: Call $P^* \leftarrow \mathsf{OProg}(P)$ and $(\tilde{P}^*, k_2^{\mathsf{in}}) \leftarrow \mathsf{GProg}(P^*, k_2, n, t^*_{init}, t^*_{cur})$ where t^*_{init}, t^*_{cur} are the updated times with the overhead of the ORAM scheme.
  Output $\tilde{P}^*, k^{\mathsf{in}} = (k_1, k_2^{\mathsf{in}})$.
- $\mathsf{GInput}'(x, k^{\mathsf{in}} = (k_1, k_2^{\mathsf{in}}))$: Output $\tilde{x}^* \leftarrow \mathsf{GInput}((x, k_1), k_2^{\mathsf{in}})$.

Theorem D.1. *If G is a garbled RAM scheme that provides UMA security and supports programs with ptWrites and O is an ORAM with ptWrites then G' is a garbled RAM with full security and supporting arbitrary programs.*

Proof. It is clear that the use of ORAM with ptWrites in the above construction ensures that G is only used on program executions that satisfy ptWrites. Therefore, we only need to prove that G' provides security. Let Sim_1 be the ORAM simulator and let Sim_2 be the simulator for G. Then we define the simulator Sim' for G' which first calls $(D^*_{\mathsf{sim}}, \mathsf{MemAccess}_{\mathsf{sim}}) \leftarrow \mathsf{Sim}_1(1^\kappa, n, \{t_i\}_{i=1}^\ell)$ to compute the simulate data and access pattern, and then outputs $\mathsf{Sim}_2(1^\kappa, \{P_i^*, t_i^*, y_i\}_{i=1}^\ell, D^*_{\mathsf{sim}}, \mathsf{MemAccess}_{\mathsf{sim}})$ where t_i^* are the updated running times after applying ORAM. Security follows from two simple hybrid arguments:

- By the security of G', the real distribution is indistinguishable from

  $$\mathsf{Sim}_2(1^\kappa, \{P_i^*, t_i^*, y_i\}_{i=1}^\ell, D^*, \mathsf{MemAccess})$$

  where $D^*, \mathsf{MemAccess}$ are the "real" data and access pattern produced by the oblivious RAM scheme.
- By the security of the ORAM scheme O, we know that $(D^*, \mathsf{MemAccess})$ is indistinguishable from the simulated $(D^*_{\mathsf{sim}}, \mathsf{MemAccess}_{\mathsf{sim}})$.

This completes the proof. □

Proof of Theorem 5.3. To prove the main theorem, Theorem 5.3, we now combine all of our results. Most importantly, combining Theorem 5.2 and Theorem D.1, we get the existence of garbled RAM with full security for arbitrary programs assuming the existence of (I) TIBE, (II) garbled circuits with wire labels, (III) ORAM with ptWrites. Known results give us ORAM and garbled circuits with wire labels (e.g., Yao) from one-way functions, and our transformation in Section A.3 shows how to convert any ORAM into one with ptWrites. Lastly Theorem B.1 shows how to construct TIBE from IBE. Since IBE implies one-way functions, we can get (I), (II) and (III) assuming the existence of IBE.