Constant-Round Maliciously Secure
Two-Party Computation in the RAM Model [*]

Carmit Hazay   Avishay Yanai

Abstract

The *random-access memory (RAM)* model of computation allows program constant-time memory lookup and is more applicable in practice today, covering many important algorithms. This is in contrast to the classic setting of secure 2-party computation (2PC) that mostly follows the approach for which the desired functionality must be represented as a boolean circuit. In this work we design the first *constant round* maliciously secure two-party protocol in the RAM model. Our starting point is the garbled RAM construction of Gentry et al. [GHL+14] that readily induces a constant round semi-honest two-party protocol for any RAM program assuming identity-based encryption schemes. We show how to enhance the security of their construction into the malicious setting while facing several challenges that stem due to handling the data memory. Next, we show how to apply our techniques to a more recent garbled RAM construction by Garg et al. [GLOS15] that is based on one-way functions.

Contents

1 Introduction

Background on secure computation. Secure multi-party computation enables a set of parties to mutually run a protocol that computes some function f on their private inputs, while preserving a number of security properties. Two of the most important properties are privacy and correctness. The former implies data confidentiality, namely, nothing leaks by the protocol execution but the computed output. The latter requirement implies that the protocol enforces the integrity of the computations made by the parties, namely, honest parties learn the correct output. More generally, a rigorous security definition requires that distrusting parties with secret inputs will be able to compute a function of their inputs as if the computation is executed in an ideal setting, where the parties send their inputs to a incorruptible trusted party that performs the computation and returns its result (also known by the ideal/real paradigm). The feasibility of secure computation has been established by a sequence of works [Yao86, GMW87, Bea91, MR91, Can00], proving security under this rigorous definition with respect to two adversarial models: the semi-honest model (where the adversary follows the instructions of the protocol but tries to learn more than it should from the protocol transcript), and the malicious model (where the adversary follows an arbitrary polynomial-time strategy).

Following these works much effort was put in order to improve the efficiency of computation with the aim of minimizing the workload of the parties [JS07, LP07, IPS08] [IPS09, PSSW09, NO09, LP11, IKO$^+$11, Lin13]. These general-purpose protocols are restricted to functions represented by Boolean/arithmetic circuits. Namely, the function is first translated into a (typically Boolean) circuit and then the protocol securely evaluates it gate-by-gate on the parties' private inputs. This approach, however, falls short when the computation involves access to a large memory since in the circuits-based approach, dynamic memory accesses, which depend on the secret inputs, are translated into a linear scan of the memory. This translation is required for every memory access and causes a huge blowup in the description of the circuit.

The RAM model of computation. We further note that the majority of applications encountered in practice today are more efficiently captured using *random-access memory (RAM)* programs that allow constant-time memory lookup. This covers graph algorithms, such as the known Dijkstra's shortest path algorithm, binary search on sorted data, finding the kth-ranked element, the Gale-Shapely stable matching algorithm and many more. This is in contrast to the sequential memory access that is supported by the architecture of Turing machines. Generic transformations from RAM programs that run in time T generate circuits of size $(T^3 \log T)$ which are non-scalable even for cases where the memory size is relatively small [CR72, PF79].

To address these limitations, researchers have recently started to design secure protocols directly in the RAM model [DMN11, GKK$^+$12, AHMR15]. The main underlying idea is to rely on Oblivious RAM (ORAM) [Gol87, Ost90, GO96], a fundamental tool that supports dynamic memory access with polylogarithmic cost while preventing any leakage from the memory. To be concrete, ORAM is a technique for hiding all the information about the memory of a RAM program. This includes both the content of the memory as well as the access pattern to it.

In more details, a RAM program P is defined by a function that is executed in the presence of memory D via a sequence of read and write operations, where the memory is viewed as an array of n entries (or blocks) that are initially set to zero. More formally, a RAM program is defined by a "next instruction" function that is executed on an input x, a current state state and data element b^{read} (that will always be equal to the last read element from memory D) and outputs the next instruction and an updated state. We use the notation $P^D(x)$ to denote the execution of such a program. To avoid trivial solutions, such as fetching the entire memory, it is required that the space used by the evaluator grows linearly with $\log n$, $|x|$ and the block length. The space complexity of a RAM program on inputs x, D is the maximum number of entries used

by P during the course of the execution. The time complexity of a RAM program on the same inputs is the number of instructions issued in the execution as described above.

Secure computation for RAM programs. An important application of ORAM is in gaining more efficient protocols for secure computation [GKK$^+$12, GGH$^+$13, LO13, GHJR14] [GHL$^+$14, KS14, LHS$^+$14, WHC$^+$14, WCS15, AHMR15, GLOS15, HMR15, GLO15]. This approach is used to securely evaluate RAM programs where the overall input sizes of the parties are large (for instance, when one of the inputs is a database). Amongst these works, only [AHMR15] addresses general secure computation for arbitrary RAM programs with security in the presence of malicious adversaries. The advantage of using secure protocols directly for RAM programs is that such protocols imply (amortized) complexity that can be sublinear in the total size of the input. In particular, the overhead of these protocols grows linearly with the time-complexity of the underlying computation on the RAM program (which may be sublinear in the input size). This is in contrast to the overhead induced by evaluating the corresponding Boolean/arithmetic circuit of the underlying computation (for which its size is linear in the input size).

One significant challenge in handling dynamic memory accesses is to hide the actual memory locations being read/written from all parties. The general approach in most of these protocols is of designing protocols that work via a sequence of ORAM instructions using traditional circuit-based secure computation phases. More precisely, these protocols are defined using two phases: (1) initialize and setup the ORAM, a one-time computation with cost depending on the memory size, (2) evaluate the next-instruction circuit which outputs shares of the RAM program's internal state, the next memory operations (read/write), the location to access, and the data value in case of a write. This approach leads to protocols with semi-honest security whom their round complexity depends on the ORAM running time. In [GKK$^+$12] Gordon et al. designed the first rigorous semi-honest secure protocols based on this approach, that achieves sublinear amortized overhead that is asymptotically close to the running time of the underlying RAM program in an insecure environment.

As observed later by Afshar et al. [AHMR15], adapting this approach in the malicious setting is quite challenging. Specifically, the protocol must ensure that the parties use state and memory shares that are consistent with prior iterations, while ensuring that the running time only depends on the ORAM running time rather than on the entire memory. They therefore consider a different approach of garbling the memory first and then propagate the output labels of these garbling within the CPU-step circuits.

The main question left open by their work is the *feasibility of constant round malicious secure computation in the RAM model*. In this work we address this question in the two-party setting.

1.1 Our Results

We design the first constant round maliciously secure protocol for arbitrary RAM programs. Our starting point is the garbled RAM construction of Gentry et al. [GHL$^+$14], which is the analogue object of garbled circuits [Yao82, BHR12] with respect to RAM programs. Namely, a user can garble an arbitrary RAM program directly without converting it into a circuit first. A garbled RAM scheme can be used to garble the data, the program and the input in a way that reveals only the evaluation outcome and nothing else. In their work, Gentry et al. proposed two ways to fix a subtle point emerged in an earlier construction by Lu and Ostrovsky [LO13] that requires a complex "circular" use of Yao garbled circuits and PRFs. For simplicity, we chose to focus on their garbled RAM based on identity based encryption (IBE) schemes. We show how to transform their IBE based protocol into a maliciously secure 2PC protocol at the cost of involving the cut-and-choose technique. Following that, we show how to achieve the same result using the more involved

garbled RAM construction of Garg et al. [GLOS15] assuming only the existence of one-way-functions. We state our main theorem below,

Theorem 1.1 (Informal). *Under the standard assumptions for achieving static malicious 2PC security, there exists a constant round protocol securely realizes any RAM program in the presence of malicious adversaries, making only black-box use of an Oblivious RAM construction, where the size of the garbled database is $|D| \cdot \mathsf{poly}(\kappa)^1$, the size of the garbled input is $|x| \cdot O(\kappa) + T \cdot \mathsf{poly}(\kappa)$ and the size of the garbled program and its evaluation time is $|\mathrm{C}_{\mathrm{CPU}}^P| \times T \times \mathsf{poly}(\kappa) \times \mathsf{polylog}(|D|) \times s$.*

Where $\mathrm{C}_{\mathrm{CPU}}^P$ is a circuit that computes a CPU-step that involves reading/writing to the memory, T is the running time of program P on input x, κ is the security parameter and s is a statistical cut-and-choose parameter.

Challenges faced in the malicious setting and RAM programs.

1. MEMORY MANAGEMENT. Intuitively speaking, garbled RAM readily induces a two-party protocol with semi-honest security by exchanging the garbled input using oblivious transfer (OT). The natural approach for enhancing the security of a garbled RAM scheme into a maliciously 2PC protocol is by using the cut-and-choose approach [LP07] where the basic underlying semi-honest protocol is repeated s times (for some statistical parameter s), such that a subset of these instances are "opened" in order to demonstrate correct behaviour whereas the rest of "unopened" instances are used to obtaining the final outcome (typically by taking the majority of results). The main challenge in boosting the security of a semi-honest secure protocol into the malicious setting, using this technique in the RAM model, is with handling multiple instances of memory data. That is, since each semi-honest protocol instance is executed independently, the RAM program implemented within this instance is associated with its own instance of memory. Recalling that the size of the memory might be huge compared to the other components in the RAM system, it is undesirable to store multiple copies of the data in the local memory of the parties. Therefore, the first challenge we had to handle is how to work with multiple copies of the same protocol while having access to a single memory data.

2. HANDLING CHECK/EVALUATION CIRCUITS. The second challenge concerns the cut-and-choose proof technique as well. The original approach to garble the memory is by using encryptions computed based on PRF keys that are embedded inside the garbled circuits. These keys are used to generate a translation mapping which allows the receiver to translate between the secret keys and the labels of the read bit in the next circuit. When employing the cut-and-choose technique, all the secret information embedded within the circuits is exposed during the check process of that procedure which might violate the privacy of the sender. The same difficulty arises when hardwiring the randomness used for the encryption algorithm. A naive solution would be to let the sender choose s sets of keys, such that each set is used within the appropriate copy of the circuit. While this solution works, it prevents the evaluator from determining the majority of the (intermediate) results of all copies.

3. INTEGRITY AND CONSISTENCY OF MEMORY OPERATIONS. During the evaluation of program P, the receiver reads and writes back to the memory. In the malicious setting these operations must be backed up with a mechanism that enforces correctness. Moreover, a corrupted evaluator should not be able to roll back the stored memory to an earlier version. This task is very challenging in a scenario

---

[1]The size mentioned is correct when relying on the IBE assumption, while relying on the OWF assumption would incur database size of $|D| \cdot \log |D| \cdot \mathsf{poly}(\kappa)$.

where the evaluator locally stores the memory and fully controls its accesses without the sender being able to verify whether the receiver has indeed carried out the required instructions (as that would imply that the round complexity grows linearly with the running time of the RAM program).

Constant round 2PC in the RAM model.   Towards achieving malicious security, we demonstrate how to adapt the garbled RAM construction from [GHL$^+$14] into the two-party setting while achieving malicious security. Our protocol is combined of two main components. First, an initialization circuit is evaluated in order to create all the IBE keys (or the PRF keys) that are incorporated in the latter RAM computation, based on the joint randomness of the parties (this phase is not computed locally since we cannot rely on the sender properly creating these keys). Next, the program P is computed via a sequence of small CPU-steps that are implemented using a circuit that takes as input the current CPU state and a bit that was read from the last read memory location, and outputs an updated state, the next location to read, a location to write to and a bit to write into that location. In order to cope with the challenges regarding the cut-and-choose approach, we must ensure that none of the secret keys nor randomness are incorporated into the circuits, but instead given as inputs. Moreover, to avoid memory duplication, all the circuits are given the same sequence of random strings. This ensures that the same set of secret keys/ciphertexts are created within all CPU circuits.

We note that our protocol is applicable to any garbled scheme that supports wire labels and can be optimized using all prior optimizations. Moreover, in a variant of our construction the initialization phase can be treated as a preprocessing phase that does not depend on the input. We further note that our abstraction of garbled circuits takes into account authenticity [BHR12]. Meaning that, a malicious evaluator should not be able to conclude the encoding of a string that is different than the actual output. This requirement is crucial for the security of garbled circuits with reusable labels (namely, where the output labels are used as input labels in another circuit), and must be addressed even in the semi-honest setting (and specifically for garbled RAM protocols). This is because authenticity is not handled by the standard privacy requirement. Yet, all prior garbled RAM constructions do not consider it. We stress that we do not claim that prior proofs are incorrect, rather that the underlying garbled circuits must adhere this security requirement in addition to privacy.

As final remark, we note that our construction employs the underlying ORAM in a black-box manner as the parties invoke it locally. This is in contrast to alternative approaches that compute the ORAM using a two (or multi)-party secure protocol such as in [GKK$^+$12].

Complexity.   The overhead of our protocol is dominated by the complexity induced by the garbled RAM construction of [GHL$^+$14] times s, where s is the cut-and-choose statistical parameter. The [GHL$^+$14] construction guarantees that the size/evaluation time of the garbled program is $|C_{\mathrm{CPU}}^P| \times T \times \mathsf{poly}(\kappa) \times \mathsf{polylog}(n)$. Therefore the multiplicative overhead of our protocol is $\mathsf{poly}(\kappa) \times \mathsf{polylog}(n) \times s$.

Reusable/persistent data.   Reusable/persistent data means that the garbled memory data can be reused across multiple program executions. That is, all memory updates are persist for future program executions and cannot be rolled back by the malicious evaluator. This feature is very important as it allows to execute a sequence of programs without requiring to initialize the data for every execution, implying that the running time is only proportional to the program running time (in a non-secured environment). The [GHL$^+$14] garbled RAM allows to garble any sequence of programs (nevertheless, this set must be given to the garbler in advance and cannot be adaptively chosen). We show that our scheme preserves this property in the presence of malicious attacks as well.

Concurrent work. In a concurrent and independent work by Garg, Gupta, Miao and Pandey [GGMP16], the authors demonstrate constant-round multi-party computation with the advantage of achieving a construction that is *black-box* in the one-way function. Their work is based on the black-box GRAM construction of [GLO15] and the constant-round MPC construction of [BMR90]. Their semi-honest secure protocol achieves persistent data, whereas their maliciously secure protocol achieves the weaker notion of selectively choosing the inputs in advance, as we do. The core technique of pulling secrets out of the programs and into the inputs is common to both our and their work. Whereas our construction achieves two features which [GLO15] does not. First, we use the ORAM in a black-box way since the parties can locally compute it. Second, only one party locally stores the memory, rather than both parties string shares of the memory. In another paper [Mia], Miao demonstrates how to achieve persistent data in the two-party setting assuming a random oracle and using techniques from [NO09] and [BHR12], where the underlying one-way function is used in a black-box manner.

2   Preliminaries

Basic Notations. We denote the security parameter by κ. We say that a function $\mu : \mathbb{N} \to \mathbb{N}$ is *negligible* if for every positive polynomial $p(\cdot)$ and all sufficiently large κ it holds that $\mu(\kappa) < \frac{1}{p(\kappa)}$. We use the abbreviation PPT to denote probabilistic polynomial-time. We further denote by $a \leftarrow A$ the random sampling of a from a distribution A, by $[d]$ the set of elements $(1, \ldots, d)$ and by $[0, d]$ the set of elements $(0, \ldots, d)$.

We now specify the definition of (κ, s)-computational indistinguishability (denoted $\overset{\kappa,s}{\approx}$), while the usual (computational indistinguishability) definition (denoted $\overset{c}{\approx}$) can be inferred.

Definition 2.1. *Let* $X = \{X(a, \kappa, s)\}_{a \in \{0,1\}^*, \kappa, s \in \mathbb{N}}$ *and* $Y = \{Y(a, \kappa, s)\}_{a \in \{0,1\}^*, \kappa, s \in \mathbb{N}}$ *be two distribution ensembles. We say that* X *and* Y *are* (κ, s)-computationally indistinguishable*, denoted* $X \overset{\kappa,s}{\approx} Y$*, if there exist a constant* $0 < c \leq 1$ *such that for every* PPT *machine* D*, every* $s \in \mathbb{N}$ *every positive polynomial* $p(\cdot)$ *and all sufficiently large* κ *it holds that for every* $a \in \{0,1\}^*$ *:*

$$\left| \Pr\left[D(X(a, \kappa), 1^\kappa) = 1 \right] - \Pr\left[D(Y(a, \kappa), 1^\kappa) = 1 \right] \right| < \frac{1}{p(\kappa)} + \frac{1}{2^{c \cdot s}}.$$

2.1   The RAM Model of Computation

We follow the notation from [GHL+14] verbatim. We consider a program P that has random-access to a memory of size n, which is initially empty. In addition, the program gets a "short" input x, which we can alternatively think of as the initial state of the program. 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. [GHL+14] also considered the case where several different programs are executed sequentially and the memory persists between executions. Our protocol follows this extension as well. Specifically, this process is denoted as $(y_1, \ldots, y_c) = (P_1(x_1), \ldots, P_\ell(x_c))^D$ 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.

CPU-step circuit. We view a RAM program as a sequence of at most T small CPU-steps, such that step $1 \leq t \leq T$ is represented by a circuit that computes the following functionality:

$$\mathrm{C}_{\mathrm{CPU}}^P(\mathsf{state}_t, b_t^{\mathsf{read}}) = (\mathsf{state}_{t+1}, i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}}).$$

Namely, this circuit takes as input the CPU state state_t and a bit b_t^{read} that was read from the last read memory location, and outputs an updated state state_{t+1}, the next location to read $i_t^{\mathsf{read}} \in [n]$, a location to write to $i_t^{\mathsf{write}} \in [n] \cap \bot$ (where $\bot$ means "write nothing") and a bit b_t^{write} to write into that location. The computation $P^D(x)$ starts in the initial state $\mathsf{state}_1 = (x_1, x_2)$, corresponding to the parties "short input" and by convention we will set the initial read bit to $b_1^{\mathsf{read}} := 0$. In each step t, the computation proceeds by running $\mathrm{C}_{\mathrm{CPU}}^P(\mathsf{state}_t, b_t^{\mathsf{read}}) = (\mathsf{state}_{t+1}, i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}})$. We first read the requested location i_t^{read} by stetting $b_{t+1}^{\mathsf{read}} := D[i_t^{\mathsf{read}}]$ and, if $i_t^{\mathsf{write}} \neq \bot$ we write to the location by setting $D[i_t^{\mathsf{write}}] := b_t^{\mathsf{write}}$. The value $y = \mathsf{state}_{T+1}$ output by the last CPU-step serves as the output of the computation.

A program P has a *read-only* memory access, if it never overwrites any values in memory. In particular, using the above notation, the outputs of $\mathrm{C}_{\mathrm{CPU}}^P$ always set $i_t^{\mathsf{write}} = \bot$.

2.1.1 Predictably Time Writes

Predictably Time Writes (ptWrites) means that whenever we want to read some location i in memory, it is easy to figure out the time (i.e., CPU step) t' in which that location was last written to, given only the current state of the computation and without reading any other values in memory. In [GHL$^+$14] the authors describe how to upgrade a solution for ptWrites to one that allows arbitrary writes. More formally,

Definition 2.2 (Predictably timed writes [GHL$^+$14])**.** *A program execution $P^D(x_1, x_2)$ has predictably timed writes if there , denoted $\mathsf{WriteTime}$, such that the following holds for every CPU step $t = 1, \ldots, T$. Let the inputs/outputs of the t-th CPU step be $\mathsf{cpu\text{-}step}(\mathsf{state}_t, b_t^{\mathsf{read}}) = (\mathsf{state}_{t+1}, i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}})$, then $t' = \mathsf{WriteTime}(t, \mathsf{state}_t, i_t^{\mathsf{read}})$ is the largest value of $t' < t$ such that the CPU step t' wrote to memory location i_t^{read}; i.e. $i_{t'}^{\mathsf{write}} = i_t^{\mathsf{read}}$.*

As in [GHL$^+$14], we also describe a solution for RAM programs that support ptWrites and then show how to extend it to the general case.

2.2 Oblivious RAM (ORAM)

ORAM, initially proposed by Goldreich and Ostrovsky [Gol87, Ost90, GO96], is an approach for making a read/write memory access pattern of a RAM program input-oblivious. More precisely, it allows a client to store private data on an untrusted server and maintain obliviousness while accessing that data, by only storing a short local state. A secure ORAM scheme not only hides the content of the memory from the server, but also the access pattern of which locations in the memory the client is reading or writing in each protocol execution.[2] The work of the client and server in each such access should be small and bounded by a poly-logarithmic factor in the memory size, where the goal is to access the data without downloading it from the server in its entirely. In stronger attack scenarios, the ORAM is also authenticated which means that the server cannot modify the content of the memory. In particular, the server cannot even "roll-back" to an older version of the data. The efficiency of ORAM constructions is evaluated by their bandwidth blowup, client storage and server storage. Bandwidth blowup is the number of data blocks that are needed to be sent between the parties per request. Client storage is the amount of trusted local memory required for the client to manage the ORAM and server storage is the amount of storage needed at the server to store all data blocks. Since the seminal sequence of works by Goldreich and Ostrovsky, ORAM has been extensively studied [SCSL11, GMOT12, KLO12, WS12, SvDS$^+$13, RFK$^+$15], optimizing different metrics and parameters.

---

[2]This can always be done by encrypting the memory.

Before giving the formal definition let us put down the settings and notations: A Random Access Machine (RAM) with memory size n consists of a CPU with a small number of registers (e.g. $poly(\kappa)$, where κ is the security parameter), that each can store a string of length κ (called a "word") and external memory of size n. A word is either $\perp$ or a κ bit string. Given n and x, the CPU executes the program P by sequentially evaluating the CPU-step function $C_{\mathrm{CPU}}^P(n, \mathsf{state}_t, b_t^{\mathsf{read}}) = (\mathsf{state}_{t+1}, i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}})$ where $t = 0, 1, 2, \ldots, T-1$ such that T is the upper bound on the program run time and $\mathsf{state}_0 = x$. The sequence of memory cells and data written in the course of the execution of the program is defined by $\mathsf{MemAccess}(P, n, x) = \{(i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}})\}_{t \in [T]}$ and the number of memory accesses that were performed during a program execution is denoted by $T(P, n, x)$ (that is, the running time of the program P with memory of size n on input x).

In this work we follow a slightly modified version of the standard definition (of [Gol87, Ost90, GO96]), in which the compiled program P^* is not hardcoded with any secret values, namely, neither secret keys for encryption/authentication algorithms nor the randomness that specifies future memory locations to be accessed by the program, rather, the compiled program obtains these secret values as input. More concretely, P^* is given two inputs: (1) a secret value k that is used to derive the keys for encrypting and authenticating the data, (2) a uniformly random string r which corresponds to the random indices that are accessed during the computation. The formal definition follows:[3]

Definition 2.3. *A polynomial time algorithm C is an Oblivious RAM (ORAM) compiler with computational overhead $c(\cdot)$ and memory overhead $m(\cdot)$, if C, when given $n \in \mathbb{N}$ and a deterministic RAM program P with memory size n, outputs a program P^* with memory size $m(n) \cdot n$, such that for any input $x \in \{0,1\}^*$, uniformly random key $k \in \{0,1\}^\kappa$ and uniformly random string $r \in \{0,1\}^\kappa$, it follows that $T(P^*(n, x, k, r)) = c(n) \cdot T(P, n, x)$ and there exists a negligible function μ such that the following properties hold:*

- **Correctness.** *For any $n \in \mathbb{N}$, any input $x \in \{0,1\}^*$, any key and uniformly random string $k, r \in \{0,1\}^\kappa$, with probability at least $1 - \mu(\kappa)$, $P^*(n, x, k, r) = P(n, x)$.*

- **Obliviousness.** *For any two programs P_1, P_2, any $n \in \mathbb{N}$, any two inputs, uniformly random keys and uniformly random strings: $x_1, x_2 \in \{0,1\}^*, k_1, k_2, r_1, r_2 \in \{0,1\}^\kappa$ respectively, if $T(P_1(n, x_1)) = T(P_2(n, x_2))$ and $P_1^* \leftarrow C(n, P_1, \rho_1)$, $P_2^* \leftarrow C(n, P_2, \rho_2)$ then $\mathsf{MemAccess}(P_1^*(n, x_1, k_1, r_1))$ and $\mathsf{MemAccess}(P_2^*(n, x_2, k_2, r_2))$ are computationally indistinguishable, where the random tapes ρ_1, ρ_2 that were used by the compiler to generate the compiled programs are given to the distinguisher.*[4]

Note that the above definition (just as the definition of [GO96]) only requires an oblivious compilation of deterministic programs P. This is without loss of generality: We can always view a randomized program as a deterministic one that receives random coins as part of its input.

2.2.1 Realization of the Modified Definition

We present here a sketch of an ORAM compiler that meets the above requirements, which is a slightly modified construction of the Simple ORAM that was presented in [CP13]. The modified compiler is a deterministic algorithm C, that is, its random tape ρ is an empty string ε. When given a program P, the

---

[3]The following definition is derived from the definition given in[CP13].

[4]The use of ρ_1, ρ_2 does not reveal any information about the access pattern nor about the encryption key of the data, these are determined only by the keys k_1, k_2 and the random strings r_1, r_2.

compiler outputs a program P^* that takes the inputs x, k, r where x is the input to the original program P, k is a uniformly random string from which the encryption/authentication keys are derived and r is a uniformly random strings of the following form: $r = \{Pos, r_1, r_2, \ldots, r_T\}$ such that Pos is the initial position map of the oblivious program and $r_1, \ldots, r_T$ are the additional random locations that are used for each iteration during the execution of the program P^*. The program P^* that C outputs is specified exactly as the the oblivious program presented in [CP13], except that the position map Pos and random paths $r_1, \ldots, r_T$ are not hardcoded within the program, rather, they are given as inputs to the program.

2.3 Secure Computation in the RAM Model

We adapt the standard definition for secure two-party computation of [Gol04, Chapter 7] for the RAM model of computation. In this model of computation, the initial input is split between two parties and the parties run a protocol that securely realizes a program P on a pair of "short" inputs x_1, x_2, which are viewed as the initial state of the program. In addition, the program P has random-access to a memory of size n which is initially empty. Using the notations from Section 2.1, we refer to this (potentially random) process by $P^D(x_1, x_2)$. In this work we prove the security of our protocols in the presence of malicious computationally bounded adversaries.

We next formalize the ideal and real executions, considering D as a common resource.[5] Our formalization induces two flavours of security definitions. In the first (and stronger) definition, the memory accesses to D are hidden, that is, the ideal adversary that corrupts the receiver only obtains (from the trusted party) the running time T of the program P and the output of computation y. Given only these inputs, the simulator must be able to produce an indistinguishable memory access pattern. In the weaker, unprotected memory access model described below, the simulator is further given the content of the memory, as well as the memory access pattern produced by the trusted party throughout the computation of P^D. We present here both definitions, starting with the definition of full security.

2.3.1 Full Security

Execution in the ideal model. In an ideal execution, the parties submit their inputs to a trusted party that computes the output; see Figure 1 for the description of the functionality computed by the trusted party in the ideal execution. Let P be a two-party program, let $\mathcal{A}$ be a non-uniform PPT machine and let $i \in \{S, R\}$ be the corrupted party. Then, denote the *ideal execution of P* on inputs (x_1, x_2), auxiliary input z to $\mathcal{A}$ and security parameters s, κ, by the random variable $\mathbf{IDEAL}^{\mathcal{F}_{\mathrm{RAM}}}_{\mathcal{A}(z),i}(s, \kappa, x_1, x_2)$, as the output pair of the honest party and the adversary $\mathcal{A}$ in the above ideal execution.

Execution in the real model. In the real model there is no trusted third party and the parties interact directly. The adversary $\mathcal{A}$ sends all messages in place of the corrupted party, and may follow an arbitrary PPT strategy. The honest party follows the instructions of the specified protocol π. Let P^D be as above and let π be a two-party protocol for computing P^D. Furthermore, let $\mathcal{A}$ be a non-uniform PPT machine and let $i \in \{S, R\}$ be the corrupted party. Then, the *real execution of π* on inputs (x_1, x_2), auxiliary input z to $\mathcal{A}$ and security parameters s, κ, denoted by the random variable $\mathbf{REAL}^{\pi}_{\mathcal{A}(z),i}(s, \kappa, x_1, x_2)$, is defined as the output pair of the honest party and the adversary $\mathcal{A}$ from the real execution of π.

---

[5]Nevertheless, we note that the memory data D will be kept in the receiver's local memory.

Functionality $\mathcal{F}_{\text{RAM}}$

The functionality $\mathcal{F}_{\text{RAM}}$ interacts with a sender S and a receiver R. The program P is known and agreed by both parties.

Input: Upon receiving input value $(\text{INPUT}_{\text{S}}, x_1)$ from S and input value $(\text{INPUT}_{\text{R}}, x_2)$ from R store x_1, x_2 and initialize the memory data D with 0^n.

Output: If both inputs are recorded execute $y \leftarrow P^D(x_1, x_2)$ and send $(\text{OUTPUT}_{\text{R}}, T, y)$ to R.

Figure 1: A 2PC secure evaluation functionality in the RAM model for program P.

Security as emulation of a real execution in the ideal model. Having defined the ideal and real models, we can now define security of protocols. Loosely speaking, the definition asserts that a secure party protocol (in the real model) emulates the ideal model (in which a trusted party exists). This is formulated by saying that adversaries in the ideal model are able to simulate executions of the real-model protocol.

Definition 2.4 (Secure computation). *Let $\mathcal{F}_{\text{RAM}}$ and π be as above. Protocol π is said to* securely compute P^D *with abort in the presence of malicious adversary if for every non-uniform PPT adversary $\mathcal{A}$ for the real model, there exists a non-uniform PPT adversary $\mathcal{S}$ for the ideal model, such that for every $i \in \{\text{S}, \text{R}\}$,*

$$\left\{ \mathbf{IDEAL}_{\mathcal{S}(z),i}^{\mathcal{F}_{\text{RAM}}}(s, \kappa, x_1, x_2) \right\}_{s, \kappa \in \mathbb{N}, x_1, x_2, z \in \{0,1\}^*} \overset{\text{c}}{\approx} \left\{ \mathbf{REAL}_{\mathcal{A}(z),i}^{\pi}(s, \kappa, x_1, x_2) \right\}_{s, \kappa \in \mathbb{N}, x_1, x_2, z \in \{0,1\}^*}$$

where s and κ are the security parameters.

We next turn to a weaker definition of secure computation in the unprotected memory access model, and then discuss a general transformation from a protocol that is secure in the UMA model to a protocol that is fully secure.

2.3.2 The UMA Model

In [GHL$^+$14], Gentry et al. considered a weaker notion of security, denoted by *Unprotected Memory Access* (UMA), in which the receiver may additionally learn the content of the memory D, as well as the memory access pattern throughout the computation including the locations being read/written and their contents.[6] In the context of two-party computation, when considering the ideal execution, the trusted party further forwards the adversary the values $\mathsf{MemAccess} = \{(i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}})\}_{t \in [T]}$ where i_t^{read} is the address to read from, i_t^{write} is the address to write to and b_t^{write} is the bit value to be written to location i_t^{write} in time step t. We denote this functionality, described in Figure 2, by $\mathcal{F}_{\text{UMA}}$. We define security in the UMA model and then discuss our general transformation from UMA to full security.

Definition 2.5 (Secure computation in the UMA model). *Let $\mathcal{F}_{\text{UMA}}$ be as above. Protocol π is said to* securely compute P^D *with UMA and abort in the presence of malicious adversaries if for every non-uniform*

---

[6]Gentry et al. further demonstrated that this weaker notion of security is useful by providing a transformation from this setting into the stronger setting for which the simulator does not receive this extra information. Their proof holds against semi-honest adversaries. A simple observation shows that their proof can be extended for the malicious 2PC setting by considering secure protocols that run the oblivious RAM and the garbling computations; see below our transformation.

PPT adversary $\mathcal{A}$ for the real model, there exists a non-uniform PPT adversary $\mathcal{S}$ for the ideal model, such that for every $i \in \{\mathrm{S}, \mathrm{R}\}$, for every $s \in \mathbb{N}, x_1, x_2, z \in \{0, 1\}^*$ and for large enough κ

$$\left\{ \mathbf{IDEAL}_{\mathcal{S}(z),i}^{\mathcal{F}_{\mathrm{UMA}}}(s, \kappa, x_1, x_2) \right\}_{s,\kappa,x_1,x_2,z} \overset{\kappa,s}{\approx} \left\{ \mathbf{REAL}_{\mathcal{A}(z),i}^{\pi}(s, \kappa, x_1, x_2) \right\}_{s,\kappa,x_1,x_2,z}$$

where s and κ are the security parameters.

Functionality $\mathcal{F}_{\mathrm{UMA}}$

The functionality $\mathcal{F}_{\mathrm{UMA}}$ interacts with a sender S and a receiver R. The program P is known and agreed by both parties.

Input: Upon receiving input value $(\mathrm{INPUT}_\mathrm{S}, x_1)$ from S and input value $(\mathrm{INPUT}_\mathrm{R}, x_2)$ from R, store x_1, x_2 and initialize the memory data D with 0^n.

Output: If both inputs are recorded, execute $y \leftarrow P^D(x_1, x_2)$ and send $(\mathrm{OUTPUT}_\mathrm{R}, T, y, \mathsf{MemAccess})$ to R, where T is the number of memory accesses that were performed during the execution, and $\mathsf{MemAccess}$ is the access pattern of the execution.


Figure 2: A 2PC secure evaluation functionality in the UMA model for program P.

2.3.3 A Transforation From UMA to Full Security

Below, we present a transformation Θ, that is given (1) a protocol π with UMA security for RAM programs that support ptWrites and (2) a secure ORAM compiler C that satisfies ptWrites,[7] and outputs a two-party protocol for arbitrary RAM programs with full security; see Figure 3 for the description of Θ. The formal theorem follows:

Theorem 2.6. *Let π be a secure two-party protocol that provides UMA security for RAM programs that support ptWrites in the presence of malicious adversaries and C an ORAM compiler that satisfies ptWrites, then Θ is a two-party protocol that provides full security for arbitrary RAM programs in the presence of malicious adversaries.*

Note that the transformation uses the ORAM compiler C and the UMA-secure protocol π in a black-box manner. In addition, the transformation preserves all the properties that are related to the memory management, i.e., the party who handles the memory in π is the same one who handles the memory in $\pi' \leftarrow \Theta(P, \pi)$. Note that the efficiency of the resulted protocol $\pi' \leftarrow \Theta(P, \pi)$ is dominated by the efficiency of the UMA-secure protocol π and the ORAM compiler C. Specifically, the recent ORAM constructions set an additional polylog overhead with respect to all relevant parameters.

Security. We next present a proof sketch to the transformation presented in Figure 3. We consider first a corrupted receiver, which is the more complicated case, and then a corrupted sender.

---

[7]As for RAM programs, ORAM schemes can also support this property. Moreover, the [GHL$^+$14] transformation discussed in Section 2.1.1 can be applied to ORAM schemes as well.

Transformation Θ from UMA to Fully Secure Protocol

Inputs: The program P that the parties wish to compute. The protocol π to compute a two-party protocol with a UMA security. A secure ORAM compiler C. The sender S has x_1 and the receiver R has x_2.

Protocol:

1. The parties generate the randomness ρ using a coin-tossing protocol.

2. The parties agree on the oblivious program $P^* \leftarrow C(P, n, \rho)$ where ρ is C's random tape.

3. The parties run $\pi(P^*)$ where R's inputs are x_1, k_1, r_1, S's inputs are x_2, k_2, r_2 and the program P^* is given n (the memory size), the parties' input $x = x_1 \| x_2$, the key and the random tape of the compiled program, i.e. $k_1 \oplus k_2$ and $r_1 \oplus r_2$, respectively. That is, the parties run $\pi(P^*(n, x_1 \| x_2, k_1 \oplus k_2, r_1 \oplus r_2))$.

Figure 3: A transformation from UMA to full security.

R is corrupted. Let $\mathcal{S}_{\mathrm{UMA}}$ be the simulator for protocol π in the UMA model. The simulator for the general model, $\mathcal{S}_{\mathrm{RAM}}$, works as follows:

1. Let T be the run time of the program P, and let $\tilde{P}$ be the program of the form:
   ```
   For i=0 To T:
       Read(k);
   ```
   for some constant $\mathtt{k} \in [n]$ (i.e. $\mathtt{k}$ is in the range of D's size). Let $\tilde{P}^* \leftarrow C(n, \tilde{P})$ and its memory access pattern $\mathsf{MemAccess}(\tilde{P}^*, n, \varepsilon)$ resulted by its execution (where ε is an empty string, since $\tilde{P}$ gets no input).

2. Given the output of the program $y = P(x, y)$, the simulator $\mathcal{S}_{\mathrm{RAM}}$ outputs the view that is the result of $\mathcal{S}_{\mathrm{UMA}}(y, \mathsf{MemAccess}(\tilde{P}^*, n, \varepsilon))$.

We claim that the view that $\mathcal{S}_{\mathrm{RAM}}$ outputs is indistinguishable from the real view of the receiver in the real execution of the protocol. Assume, by contradiction, that there exist inputs x_1, x_2 for which there exists a distinguisher $\mathcal{D}$ who can distinguish between the two views with more than negligible probability. Consider the following hybrid view **Hyb** which is constructed as follows: Given P, x_1, x_2, n, compute $P^* \leftarrow C(n, P)$, choose random strings k and r and run $P^*(n, x_1 \| x_2, k, r)$. Let $\mathsf{MemAccess}(P^*, n, (x_1 \| x_2, k, r))$ be the access pattern induced by this execution, then, outputs $\mathbf{Hyb} = \mathcal{S}_{\mathrm{UMA}}(y, \mathsf{MemAccess}(P^*, n, (x_1 \| x_2, k, r)))$. The indistinguishability between $\mathcal{S}_{\mathrm{RAM}}(y)$ and $\mathbf{Hyb}(P, x_1, x_2, n)$ is reduced to the obliviousness of the ORAM compiler and the indistinguishability between $\mathbf{Hyb}(P, x_1, x_2, n)$ and the real view is reduced to the indistinguishability of the simulation of $\mathcal{S}_{\mathrm{UMA}}$ and the real view of the execution of protocol π.

S is corrupted. This case is simpler since, by the definitions of functionalities $\mathcal{F}_{\mathrm{UMA}}$ and $\mathcal{F}_{\mathrm{RAM}}$ the sender receives no output from the computation, thus, the same simulator used in the UMA model works in the general RAM model, that is $\mathcal{S}_{\mathrm{RAM}} = \mathcal{S}_{\mathrm{UMA}}$. Specifically, indistinguishability between the output of $\mathcal{S}_{\mathrm{RAM}}$ and the sender's view in the real execution in the RAM model is immediately reduced to the indistinguishability between the output of $\mathcal{S}_{\mathrm{RAM}}$ and the view in the real execution in the UMA model.

Note that our ORAM compiler definition simplifies the transformation to full security as now the result oblivious program P^* gets its randomness r as part of its input, rather than being hardcoded with it. Furthermore, recalling that this randomness is used to determine the future locations in memory for which the oblivious program is going to access, we note that r is not revealed as part of the "check circuits" when using the cut-and-choose technique.

2.3.4 On the Capabilities of Semi-Honest in a Garbled RAM and ORAM Schemes

When considering ORAM schemes in the context of two-party computation, it must be ensured that a read operation is carried out correctly. Namely, that the correct element from the memory is indeed fetched, and that the adversary did not "roll back" to an earlier version of that memory cell. Importantly, this is not just a concern in the presence of malicious adversaries, as a semi-honest adversary may try to execute its (partial) view on inconsistent memory values. Therefore, the scheme must withhold such attacks. Handling the first attack scenario is relatively simply using message authentications codes (MACs), so that a MAC tag is stored in addition to the encrypted data. Handling roll backs is slightly more challenging and is typically done using Merkle trees. In [GHL+14] roll backs are prevented by creating a new secret key for each time period. This secret key is used to decrypt a corresponding ciphertext in order to extract the label for the next garbled circuit. By replacing the secret key each time period, the adversary is not able decrypt a ciphertext created in some time period with a secret key that was previously generated.

2.4 Timed IBE [GHL+14]

TIBE was introduced by Gentry et al. in [GHL+14] in order to handle memory data writings in their garbled RAM construction. This primitive allows to create "time-period keys" TSK_t for arbitrary time periods $t \geq 0$ such that TSK_t can be used to create identity-secret-keys $\mathsf{SK}_{(t,v)}$ for identities of the form (t, v) for arbitrary v but cannot break the security of any other identities with $t' \neq t$. Gentry et al. demonstrated how to realize this primitive based on IBE [BF03, BB11]. Informally speaking, the security of TIBE is as follows: Let t^* be the "current" time period. Given a single secret key $\mathsf{SK}_{(t,v)}$ for every identity (t, v) of the "past" periods $t < t^*$ and a single period key TSK_t for every "future" periods $t^* < t \leq T$, semantic security should hold for any identity of the form $\mathsf{id}^* = (t^*, v^*)$ (for which neither a period nor secret key were not given). The formal definition of Timed IBE which is used across our protocol is as follows:[8]

Definition 2.7 (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.

- $\mathsf{TSK}_t \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, t)$: *Generates a time-period key for time-period* $t \in \mathbb{N}$.

- $\mathsf{sk}_{(t,v)} \leftarrow \mathsf{KeyGen}(\mathsf{TSK}_t, (t, v))$: *creates a secret key for the identity* (t, v).

- $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((t, v), \mathsf{msg})$: *creates an encryption of* msg *under the identity* (t, v).

- $\mathsf{msg} = \mathsf{Dec}_{\mathsf{sk}_{(t,v)}}(\mathsf{ct})$: *decrypts a ciphertexts* ct *for the identity* (t, v) *using a secret key* $\mathsf{sk}_{(t,v)}$.

The scheme should satisfy the following properties:

---

[8]We omit from the following definition the multiple secret keys that the adversary receives for identities of the form $(0, v)$ since in our scheme, data initialization is done as part of the computation if required.

Correctness: For any $\mathsf{id} = (t, v)$, and any $\mathsf{msg} \in \{0,1\}^*$ it holds that:

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

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

- The attacker $\mathcal{A}(1^\kappa)$ chooses some identity $\mathsf{id}^* = (t^*, v^*)$ with $t^* \in \mathbb{N}$ and some bound $T \geq t^*$ (given in unary). The attacker also chooses a set of identities I such that I contains exactly one identity (t, v) for each period $t \in 1, \ldots t^* - 1$. 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 $\mathsf{TSK}_t \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, t)$ for $t \in [T]$. For each $\mathsf{id} = (t, v) \in I$ it chooses $\mathsf{sk}_{\mathsf{id}} \leftarrow \mathsf{KeyGen}(\mathsf{TSK}_t, \mathsf{id})$. 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:

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

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

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

2.5 Garbled RAM Based on IBE [GHL$^+$14]

Our starting point is the garbled RAM construction of [GHL$^+$14]. Intuitively speaking, garbled RAM [LO13] is an analogue object of garbled circuits [Yao82, BHR12] with respect to RAM programs. The main difference when switching to RAM programs is the requirement of maintaining a memory data D. In this scenario, the data is garbled once, while many different programs are executed sequentially on this data. As pointed out in the modeling of [GHL$^+$14], the programs can only be executed in the specified order, where each program obtains a state that depends on prior executions. The [GHL$^+$14] garbled RAM proposes a fix to the aforementioned circularity issue raised in [LO13] by using an Identity Based Encryption (IBE) scheme [BF03, BB11] instead of a symmetric-key encryption scheme.

In more details, the inputs D, P, x to the garbled RAM are garbled into $\tilde{D}, \tilde{P}, \tilde{x}$ such that the evaluator reveals the output $\tilde{P}(\tilde{D}, \tilde{x}) = P(D, x)$ and nothing else. A RAM program P with running time T can be evaluated using T copies of a Boolean circuit $\mathrm{C}^P_{\mathrm{CPU}}$ where $\mathrm{C}^t_{\mathrm{CPU}}$ computes the function $\mathrm{C}^P_{\mathrm{CPU}}(\mathsf{state}_t, b^{\mathsf{read}}_t) = (\mathsf{state}_{t+1}, i^{\mathsf{read}}_t, i^{\mathsf{write}}_t, b^{\mathsf{write}}_t)$. Then secure evaluation of P is possible by having the sender S garble the circuits $\{\mathrm{C}^t_{\mathrm{CPU}}\}_{t \in [T]}$ (these are called the garbled program $\tilde{P}$), whereas the receiver R sequentially evaluates these circuits. In order for the evaluation to be secure the state of the program should remain secret when moving from one circuit to another. To this end, the garbling is done in a way that assigns the output wires of one circuit with the same labels as the input wires of the next circuit. The main challenge here is to preserve the ability to read and write from the memory while preventing the evaluator from learning anything beyond the program's output, including any intermediate value.

The original idea from [LO13] employed a clever usage of a PRF for which the secret key is embedded inside all the CPU-steps circuits, where the PRF's role is twofold. For reading from the memory it is used to produce ciphertexts encrypting the labels of the input wire of the input bit of the next circuit, whereas for writing it is used to generate secret keys for particular "identities". As explained in [GHL$^+$14], the proof of [LO13] does not follow without assuming an extra circularity assumption. In order to avoid circularity,

Gentry et al. proposed to replace the PRF with a public-key primitive. As it is insufficient to use a standard public-key cryptosystem (since the circuit must still produce secret keys for each memory location i, storing the keys $\mathsf{sk}_{i,0}, \mathsf{sk}_{i,1}$), the alternative is to use IBE. Below, we briefly describe their scheme.

The read-only solution. The initialized garbled data $\tilde{D}$ contains a secret key $\mathsf{sk}_{i,b}$ in each memory location $i \in [n]$ where $D[i] = b$, such that i, b serves as an identity secret key for the "identity" (i, b). Moreover, each garbled circuit $\mathrm{GC}_{\mathrm{CPU}}^t$ is hardwired with the master public key MPK of an IBE scheme.[9] This way, the garbled circuit can encrypt the input labels for the next circuit, that are associated with the bit that has just been read from the memory. More specifically, the circuit generates two ciphertexts $\mathsf{ct}_0, \mathsf{ct}_1$ that are viewed as a translation map. Namely, $\mathsf{ct}_b = \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id} = (i, b); \mathsf{msg} = \mathsf{lbl}_b^{t+1})$ and the correct label is extracted by decrypting the right ciphertext using $\mathsf{sk}_{i,b}$, such that $\mathsf{lbl}_0^{t+1}, \mathsf{lbl}_1^{t+1}$ are the input labels in the next garbled circuit that are associated with the last input bit read from the memory.

The read-write solution. A complete solution that allows both reading and writing is slightly more involved. We describe how to garble the data and the program next.

GARBLING THE DATA. The garbled data consists of secret keys $\mathsf{sk}_{(t,i,b)}$ for identities of the form $\mathsf{id} = (t, i, b)$ where i is the location in the memory D', t is the last time step for which that location was written to and $b \in \{0, 1\}$ is the bit that was written to location i at time step t. The honest evaluator only needs to keep the the most recent secret key for each location i.

GARBLING THE PROGRAM. Next, each CPU garbled circuit computes the last time step in which memory location i was written to by computing $t' = \mathsf{WriteTime}(t, \mathsf{state}, i_t^{\mathsf{read}})$. Namely, if at time step t the garbled circuit $\mathrm{GC}_{\mathrm{CPU}}^t$ instructs to read from location i_t^{read}, then the circuit further computes the last time step, u, in which that i_t^{read} was written to, it then computes the translation map $\mathsf{translate}_t = (\mathsf{ct}_0, \mathsf{ct}_1)$ by $\mathsf{ct}_b = \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id} = (u, i_t^{\mathsf{read}}, b); \mathsf{msg} = \mathsf{lbl}_b^{t+1}))$, and outputs it in the clear.

In order to write at time step t to memory location $i = i_t^{\mathsf{write}}$ the value $b = b_t^{\mathsf{write}}$, a naive solution would hardwire MSK within each garbled circuit and then generate the key $\mathsf{sk}_{(t,i,b)} = \mathsf{KeyGen}_{\mathsf{MSK}}(\mathsf{id} = (t, i, b))$; but this idea re-introduces the circularity problem. Instead, Gentry et al. [GHL$^+$14] solve this problem by introducing a new primitive called Timed IBE (TIBE). Informally, this is a two-level IBE scheme in which the first level includes the master public/secret keys (MPK, MSK) whereas the second level has T timed secret keys $\mathsf{TSK}_1, \ldots, \mathsf{TSK}_T$. The keys MPK, MSK are generated by $\mathsf{MasterGen}(1^\kappa)$ and the timed keys are generated by $\mathsf{TSK}_t = \mathsf{TimeGen}(\mathsf{MSK}, t)$.

Then in the garbling phase, the key TSK_t is hardwired within the tth garbled circuit $\mathrm{GC}_{\mathrm{CPU}}^t$ and is used to write the bit b_t^{write} to memory location i_t^{write}. To do that $\mathrm{GC}_{\mathrm{CPU}}^t$ computes the secret key for identity (t, i, b) by $\mathsf{sk}_{(t,i,b)} \leftarrow \mathsf{KeyGen}(\mathsf{TSK}_t, (t, i, b))$ which is then stored in memory location i by the evaluator. Note that $\mathrm{GC}_{\mathrm{CPU}}^t$ outputs a secret key for only one identity in every time step (for (t, i, b) but not $(t, i, 1 - b)$). This solution bypasses the circularity problem since the timed secret keys TSK_t are hardwired only within the garbled circuit computing $\mathrm{C}_{\mathrm{CPU}}^t$, and cannot be determined from either $\mathsf{sk}_{(t,i,b)}$ or the garbled circuit, provided that the TIBE scheme and the garbling schemes are secure.

2.6 Garbled Circuits

The idea of garbled circuit is originated in [Yao82]. Here, a sender can encode a Boolean circuit that computes some PPT function f, in a way that (computationally) hides from the receiver any information

---

[9] For ease of presentation, Gentry et al. abstract the security properties of the IBE scheme using a new primitive denoted by Timed IBE (TIBE); see Section 2.4 for more details.

but the function's output. In this work we consider a variant of the definition from [GHL$^+$14] that abstracts out the security properties of garbled circuits that are needed via the notion of a *garbled circuit with wire labels*. The definition that we propose below stems from the cut-and-choose technique chosen to deal with a malicious sender. Specifically, the sender uses the algorithm Garb to generate s garbled versions of the circuit C, namely $\{\tilde{C}_i\}_{i\in[s]}$ for some statistical parameter s. Then, in order to evaluate these circuits the sender sends $\{\tilde{C}_i\}_{i\in[s]}$ along with the garbled inputs $\{\tilde{x}_i\}_{i\in[s]}$, such that $\tilde{x}_i$ is the garbled input for the garbled circuit $\tilde{C}_i$. The evaluator then chooses a subset $Z \subset s$ and evaluates the garbled circuits indexed by $z \in Z$ using algorithm Eval. Note that in the notion of garbled circuits with wire labels the garbled inputs $\tilde{x}_i$ are associated with a single label per input wire of the circuit $\tilde{C}_i$; we denote these labels by $\tilde{x}_i = (\mathsf{lbl}^{1,i}_{\mathrm{in},x[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{in}},i}_{\mathrm{in},x[v_{\mathrm{in}}]})$ (where v_{in} is the number of input wires in C and $x = x[1], \ldots, x[v_{\mathrm{in}}]$ is the input to the circuit). The evaluator learns s sets[10] of output-wire labels $\{\tilde{y}_i\}_{i\in[s]}$ corresponding to the output $y = \mathrm{C}(x)$[11], where $\tilde{y}_i = (\mathsf{lbl}^{1,i}_{\mathrm{out},y[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{out}},i}_{\mathrm{in},y[v_{\mathrm{out}}]})$, but nothing else (for example, it does not learn $\mathsf{lbl}^{1,i}_{\mathrm{out},1-y[1]}$). For clarity, in the following exposition the label $\mathsf{lbl}^{j,i}_{\mathrm{in},b}$ is the label that represents the bit-value $b \in \{0,1\}$ for the jth input wire ($j \in v_{\mathrm{in}}$) in the ith garbled version of the circuit (for $i \in s$), namely $\tilde{C}_i$. Analogously, $\mathsf{lbl}^{j,i}_{\mathrm{out},b}$ represents the same, except that it is associated with an output wire (where $j \in v_{\mathrm{out}}$).

We further abstract two important properties of *authenticity* and *input consistency*. Loosely speaking, the authenticity property ensures that a malicious evaluator will not be able to produce a valid encoding of an incorrect output given the encoding of some input and the garbled circuit. This property is required due to the reusability nature of our construction. Namely, given the output labels of some iteration, the evaluator uses these as the input labels for the next circuit. Therefore, it is important to ensure that it cannot enter an encoding of a different input (obtained as the output from the prior iteration). In the abstraction used in our work, violating authenticity boils down to the ability to generate a set of output labels that correspond to an incorrect output. Next, a natural property that a maliciously secure garbling scheme has to provide is *input consistency*. We formalize this property via a functionality, denoted by $\mathcal{F}_{\mathrm{IC}}$. That is, given a set of garbled circuits $\{\tilde{C}_i\}_i$ and a set of garbled inputs $\{\tilde{x}_i\}_i$ along with the randomness r that was used by Garb; the functionality outputs 1 if the s sets of garbled inputs $\{\tilde{x}_i\}_{i=1}^s$ (where $|\tilde{x}_i| = j$) represent the same input value, and 0 otherwise. This functionality is described in Figure 7. We now proceed to the formal definition.

Definition 2.8 (Garbled circuits.)**.** *A circuit garbling scheme with wire labels consists of the following two polynomial-time algorithms:*

- *The garbling algorithm* Garb*:*

$$\left(\{\tilde{C}_i\}_i, \{u, b, \mathsf{lbl}^{u,i}_{\mathrm{in},b}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, \mathrm{C}, \{v, b, \mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b}\right)$$

for every $u \in [v_{\mathrm{in}}], v \in [v_{\mathrm{out}}], i \in [s]$ and $b \in \{0,1\}$. That is, given a circuit C with input size v_{in}, output size v_{out} and s sets of output labels $\{v, b, \mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b}$, outputs s garbled circuits $\{\tilde{C}_i\}_{i\in[s]}$ and s sets of input labels $\{u, b, \mathsf{lbl}^{u,i}_{\mathrm{in},b}\}_{u,i,b}$.

- *The evaluation algorithm* Eval*:*

$$\left\{\mathsf{lbl}^{1,i}_{\mathrm{out}}, \ldots, \mathsf{lbl}^{v_{\mathrm{out}},i}_{\mathrm{out}}\right\}_{i\in[s]} = \mathsf{Eval}\left(\{\tilde{C}_i, (\mathsf{lbl}^{1,i}_{\mathrm{in}}, \ldots, \mathsf{lbl}^{v_{\mathrm{in}},i}_{\mathrm{in}})\}_{i\in[s]}\right).$$

---

[10]This s might be different from the s used in the garbling algorithm, still we used the same letter for simplification.
[11]Note that this holds with overwhelming probability since some of the garbled circuits might be malformed.

That is, given s garbled circuits $\{\tilde{C}_i\}_i$ and s sets of input labels $\{\mathsf{lbl}_{\mathrm{in}}^{1,i}, \ldots, \mathsf{lbl}_{\mathrm{in}}^{v_{\mathrm{in}},i}\}_i$, outputs s sets of output labels $\{\mathsf{lbl}_{\mathrm{out}}^{1,i}, \ldots, \mathsf{lbl}_{\mathrm{out}}^{v_{\mathrm{out}},i}\}_i$. Intuitively, if the input labels $(\mathsf{lbl}_{\mathrm{in}}^{1,i}, \ldots, \mathsf{lbl}_{\mathrm{in}}^{v_{\mathrm{in}},i})$ correspond to some input $x \in \{0,1\}^{v_{\mathrm{in}}}$ then the output labels $(\mathsf{lbl}_{\mathrm{out}}^{1,i}, \ldots, \mathsf{lbl}_{\mathrm{out}}^{v_{\mathrm{out}},i})$ should correspond to $y = C(x)$.

Furthermore, the following properties hold.

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

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

where $\left(\{\tilde{C}_i\}_i, \{u, b, \mathsf{lbl}_{\mathrm{in},b}^{u,i}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, C, \{v, b, \mathsf{lbl}_{\mathrm{out},b}^{v,i}\}_{v,i,b}\right)$ as described above.

Verifying the correctness of a circuit. Note that in a cut-and-choose based protocols, the receiver is instructed to check the correctness of a subset of the garbled circuits. This check can be accomplished by the sender sending the receiver the randomness used in Garb. In our protocol this is accomplished by giving the receiver *both* input labels for each input wire of the check circuits, for which it can verify that the circuit computes the agreed functionality. We note that this check is compatible with all prior known garbling schemes.

Privacy. For privacy, we require that there is a PPT simulator SimGC such that for any C, x, Z and $\{\mathsf{lbl}_{\mathrm{out}}^{1,z}, \ldots, \mathsf{lbl}_{\mathrm{out}}^{v_{\mathrm{out}},z}\}_{z \in [Z]}, \{v, b, \mathsf{lbl}_{\mathrm{out},b}^{v,z}\}_{v,z \notin [Z],b}$ (i.e. one output label for wires in circuits indexed by $z \notin Z$ and a pair of output labels for wires in circuits indexed by $z \in Z$), we have

$$\left(\{\tilde{C}_z \, (\mathsf{lbl}_{\mathrm{in},x[1]}^{1,z}, \ldots, \mathsf{lbl}_{\mathrm{in},x[v_{\mathrm{in}}]}^{v_{\mathrm{in}},z})\}_z\right) \overset{c}{\approx} \mathsf{SimGC}\left(1^\kappa, \{\mathsf{lbl}_{\mathrm{out}}^{1,z}, \ldots, \mathsf{lbl}_{\mathrm{out}}^{v_{\mathrm{out}},z}\}_{z \in [Z]}, \{v, b, \mathsf{lbl}_{\mathrm{out},b}^{v,z}\}_{v,i \notin [Z],b}\right)$$

where $\left(\{\tilde{C}_z\}_z, \{u, b, \mathsf{lbl}_{\mathrm{in},b}^{u,z}\}_{u,z,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, C, \{v, b, \mathsf{lbl}_{\mathrm{out},b}^{v,z}\}_{v,z,b}\right)$ and $y = C(x)$.

Authenticity. We describe the authenticity game in Figure 8 (Appendix A.1) where the adversary is obtained a set of garbled circuits and garbled inputs for which the adversary needs to output a valid garbling of an invalid output. Namely, a garbled scheme is said to have *authenticity* if for every circuit C, for every PPT adversary $\mathcal{A}$, every s and for all large enough κ the probability $\Pr[\mathrm{Auth}_{\mathcal{A}}(1^\kappa, s, C) = 1]$ is negligible. Our definition is inspired by the definition from [BHR12] and also adapted for the cut-and-choose approach.

Input Consistency. We abstract out the functionality that checks the validity of the sender's input across all garbled circuits. We say that, a garbling scheme has *input consistency* (in the context of cut-and-choose based protocols) if there exists a protocol that realize the $\mathcal{F}_{\mathrm{IC}}$ functionality described in Figure 7 (Appendix A.1).

Realizations of our garbled circuits notion. We require the existence of a protocol Π_{IC} that securely realizes the functionality $\mathcal{F}_{\mathrm{IC}}$ described in Figure 7, in the presence of malicious adversaries. In Appendix B we exemplify this realization with [LP11].

3 Building Blocks

In this section we show how to overcome the challenges discussed in the introduction and design the first maliciously secure 2PC protocol that does not require duplication of the data and works for every garbling scheme that supports our definition based on wire labels. Recall first that in [GHL$^+$14] Gentry et al. have used a primitive called Timed IBE, where the secret-key for every memory location and stored bit (i, b) is enhanced with another parameter: the last time step t in which it has been written to the memory. The secret-key $\mathsf{sk}_{(t,i,b)}$ for identity $\mathsf{id} = (t, i, b)$ is then generated using the hard-coded time secret-key TSK_t. Now, since algorithm KeyGen is randomized, running this algorithm s times will yield s independent secret timed keys. This results in s different values to be written to memory at the same location, which implies duplication of memory data D. In order to avoid this, our solution forces the s duplicated garbled circuits for time step t to use the same random string r, yielding that all garbled circuits output the same key for the identity (t, i, b). Importantly, this does not mean that we can hard-code r in all those s circuits, since doing this would reveal r when applying the cut-and-choose technique on these garbled circuits as half of the circuits are opened. Clearly, we cannot reveal the randomness to the evaluator since the security definition of IBE (and Timed IBE) does not follow in such a scenario. Instead, we instruct the sender to input the *same randomness* in all s copies of the circuits and then run an input consistency check to these inputs in order to ensure that this is indeed the case. We continue with describing the components we embed in our protocol. An overview of the circuits involved in our protocol can be found in Figure 4 and a high-level overview of our protocol can be found in Section 4.
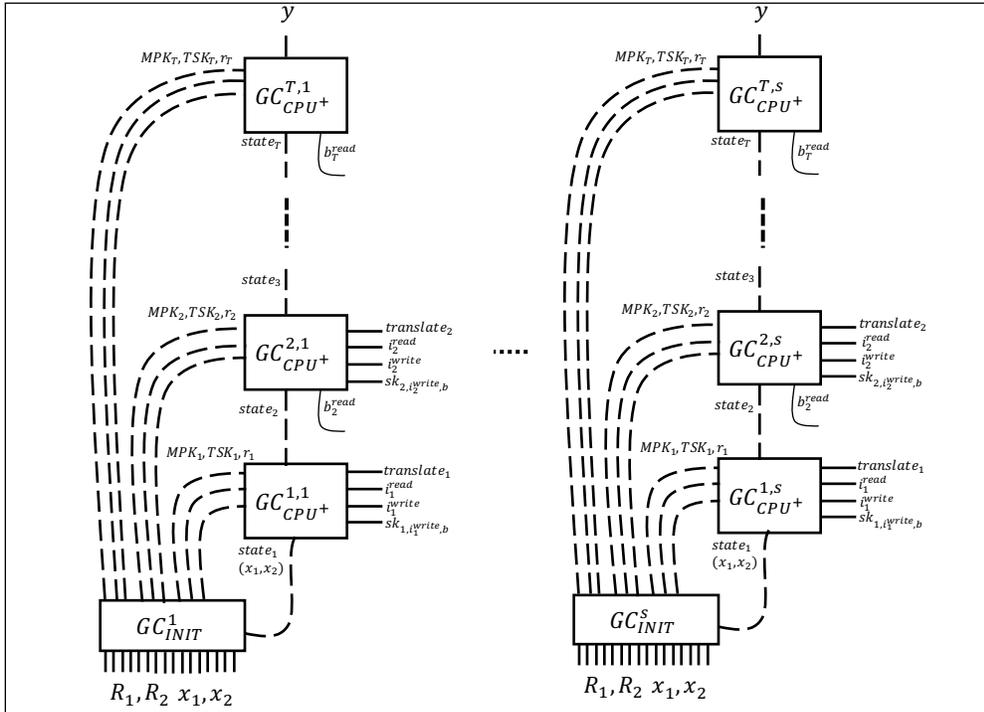
Figure 4: Garbled chains $\mathrm{GC}_{\mathrm{INIT}}, \mathrm{GC}_{\mathrm{CPU+}}^{1,i}, \ldots \mathrm{GC}_{\mathrm{CPU+}}^{T,i}$ for $i \in [s]$. Dashed lines refer to values that are passed privately (as one label per wire) whereas solid lines refer to values that are given in the clear.

3.1 Enhanced CPU-Step Function

The enhanced cpustep$^+$ function is essentially the CPU-step functionality specified in Section 2.1 enhanced with more additional inputs and output, and defined as follows

$$\text{cpustep}^+(\text{state}_t, b_t^{\text{read}}, \text{MPK}, \text{TSK}_t, r_t) = (\text{state}_{t+1}, i_t^{\text{read}}, i_t^{\text{write}}, b_t^{\text{write}}, \text{translate}_t)$$

where the additional inputs MSK, TSK_t and r_t are the master public-key, a timed secret-key for time t and the randomness r used by the KeyGen algorithm. The output translate_t is a pair of ciphertexts ct_1, ct_2, encrypted under MPK, that allows the evaluator to obtain the appropriate label of the wire that corresponds to the input bit in the next circuit. We denote the circuit that computes that function by $\text{C}_{\text{CPU}+}^t$. The functionality of $\text{C}_{\text{CPU}+}^t$ is described in Figure 5. We later describe how to securely realize this function and, in particular, how these new additional inputs are generated and given to the T CPU-circuits. The enhanced CPU-step circuit wraps the WriteTime algorithm defined in Definition 2.2.

3.2 Initialization Circuit

The initialization circuit generates all required keys and randomness to our solution and securely transfer them to the CPU-step circuits. As explained before, our solution requires the parties to input not only their input to the program but also a share to a randomness that the embedded algorithms would be given (that is, the randomness is not fixed by one of the parties). The circuit is described in figure 6.

3.3 Batch Single-Choice Cut-And-Choose OT

As a natural task in a cut-and-choose based protocol, we need to carry out cut-and-choose oblivious transfers for all wires in the circuit, for which the receiver picks a subset $Z \subset [s]$ and then obtains either both input labels (for circuits indexed with $z \in Z$), or the input label that matches the receiver's input otherwise. It is crucial that the subset of indices for which the receiver obtains both input labels is the same in all transfers. The goal of this functionality is to ensure the input consistency of the receiver and it is named by "batch single-choice cut-and-choose OT" in [LP11]. See Figure 9 (Appendix A.3) for its formal definition.

In addition to the above, our protocol uses the following building blocks: A garbling scheme $\pi_{\text{GC}} = (\text{Garb}, \text{Eval})$ that preserves the security properties from Definition 2.8; Timed IBE scheme (Section 2.4) $\pi_{\text{TIBE}} = (\text{MasterGen}, \text{TimeGen}, \text{KeyGen}, \text{Enc}, \text{Dec})$ with security as specified in Definition 2.7 and a statistically binding commitment scheme Com.

4 The Complete Protocol

Given the building blocks detailed in Section 3, we are now ready to introduce our complete protocol. Our description incorporates ideas from both [LP11] and [GHL$^+$14]. Specifically, we borrow the cut-and-choose technique and the cut-and-choose OT abstraction from [LP11] (where the latter tool enables to ensure input consistency for the *receiver*). Moreover, we extend the garbled RAM ideas presented in [GHL$^+$14] for a maliciously secure two-party protocol in the sense that we modify their garbled RAM to support the cut-and-choose approach. This allows us to obtain constant round overhead. Before we delve into the details of the protocol, let us present an overview of its main steps:

The parties wish to run the program P on inputs x_1, x_2 with the aid of an external random access storage D. In addition to their original inputs, the protocol instructs the parties to provide random strings R_1, R_2 that suffice for all the randomness needed in the execution of the CPU step circuits.

<div style="border: 1px solid black; padding: 10px;">

Enhanced CPU-Step Circuit C_{CPU+}^t

This circuit computes the enhanced CPU-step function $cpustep^+$. This circuit wraps the following algorithms: (1) the usual cpu-step for computing the next CPU-step of program P, (2) WriteTime which computes the last time t' that the program wrote to location i_t^{read} and (3) the TIBE related functionalities KeyGen and Enc. Furthermore, the labels lbl_0^{t+1} and lbl_1^{t+1} are hard coded in the circuit.

Inputs.

- $state_t$ - the last state that was output by the previous circuit. We define $state_1$ to be the parties' inputs x_1, x_2 and set b_0^{read} to be zero.

- b_t^{read} - the last bit that was read from the memory data (i.e. b_t^{read} was read from location i_t^{read}).

- MPK - the master public key of the TIBE scheme.

- TSK_t - a timed secret-key.

- r_t - randomness to be used by algorithms KeyGen and Enc_{MPK}.

Outputs. C_{CPU+}^t invokes C_{CPU}^t (the usual CPU-step circuit) that computes:

$$cpu\text{-}step(state_t, b_t^{read}) = (state_{t+1}, i_t^{read}, i_t^{write}, b_t^{write})$$

where $state_{t+1}$ is the next state of the program; i_{t+1}^{read} is the next location to read from; i_{t+1}^{write} is the next location to write to and b_{t+1}^{read} is the bit to write to location i_{t+1}^{write}.
The circuit outputs the translation $translate_t = (ct_t^0, ct_t^1)$ defined by:

$$
\begin{aligned}
t' &= WriteTime(i_{t+1}^{read}) \\
ct_t^0 &= Enc_{MPK}(id = (t, t', 0), msg = lbl_{t+1}^0) \\
ct_t^1 &= Enc_{MPK}(id = (t, t', 1), msg = lbl_{t+1}^1)
\end{aligned}
$$

Finally, the circuit computes $sk_{(t,i,b)} = KeyGen(TSK_t, id = (t, i_{t+1}^{write}, b_{t+1}^{write}))$ and outputs

$$(state_{t+1}, i_{t+1}^{read}, i_{t+1}^{write}, sk_{(t,i,b)}, translate_t).$$

</div>

Figure 5: The CPU-step circuit.

- **Chains construction.** Considering a sequence of circuits $C_{INIT}, C_{CPU+}^1, \ldots, C_{CPU+}^T$ as a *connected chain of circuits*, the sender S first generates s versions of garbled chains $GC_{INIT}^i, GC_{CPU+}^{1,i}, \ldots, GC_{CPU+}^{T,i}$ for every $i \in [s]$. It does so by iteratively feeding the algorithm Garb with s sets of pairs of output labels, where the first set of output labels lbl_{out} are chosen uniformly and are fed, together with the circuit C_{CPU+}^T, to procedure Garb, which in turn, outputs s sets of input labels. This process is being repeated till the first circuit in the chain, i.e C_{INIT}, the last s sets of input labels are denoted lbl_{in}.

- **Cut-and-choose.** Then, the parties run the batch Single-Choice Cut-and-choose OT protocol Π_{SCCOT} on the receiver's input labels, which let the receiver obtain a pair of labels for each of its input wires for every *check chain* with an index in $Z \subset [s]$ and a single label for each of its input wires for the *evaluation chains* with an index not in Z, where Z is input by the receiver to Π_{SCCOT}.

Initialization Circuit C_{INIT}

The circuit generates all keys and randomness for the T CPU step circuits $C^1_{\text{CPU}+}, \ldots, C^T_{\text{CPU}+}$.

Inputs.

- The parties input x_1, x_2, and

- $(2 \cdot (1 + T + T + 2T)) \cdot m = (8T + 2) \cdot m$ random values where m an upper bound on the length of the randomness required to run the TIBE algorithms: MasterGen, TimeGen, KeyGen and Enc. This particular number of random values is explained below.

Computation. Let R_1 (resp. R_2) be the first (resp. last) $(4t + 1) \cdot m$ bits of the inputs for the randomness. The circuit computes $R = R_1 \oplus R_2$ and interprets the result $(4t + 1) \cdot m$ bits as follows: (each of the following is a m-bit string)

- $r^{\text{MasterGen}}$ used to generate the keys MPK and MSK.

- $r_1^{\text{TimedGen}}, \ldots, r_T^{\text{TimedGen}}$ used to generate the timed secret-keys $\mathsf{TSK}_1, \ldots, \mathsf{TSK}_T$.

- $r_1^{\text{KeyGen}}, \ldots, r_T^{\text{KeyGen}}$ used to generate secret-keys $\{\mathsf{sk}_{t,i,b}\}_{t \in [T], i \in [n], b \in \{0,1\}}$ written to memory.

- $\{r_{t,b}^{\text{Enc}}\}_{t \in [T], b \in \{0,1\}}$ are used by the encryption algorithm within the CPU circuits. (Recall that the tth enhanced CPU step circuit $C^t_{\text{CPU}+}$ encrypts the two labels of the input wire that corresponds to the input bit of the next circuit $C^{t+1}_{\text{CPU}+}$.)

Then, the circuit computes:

$$(\mathsf{MPK}, \mathsf{MSK}) = \mathsf{MasterGen}(1^\kappa; r^{\text{MasterGen}})$$
$$\forall_{t \in [T]} : \mathsf{TSK}_t = \mathsf{TimeGen}(\mathsf{MSK}, t; r_t^{\text{TimedGen}})$$

Outputs.

$$\left(x_1, x_2, \{\mathsf{MPK}_t\}_{t \in [T]}, \{\mathsf{TSK}_t\}_{t \in [T]}, \{r_t^{\mathsf{KeyGen}}\}_{t \in [T]}, \{r_{t,b}^{\mathsf{Enc}}\}_{t \in [T], b \in \{0,1\}} \right)$$

where $\mathsf{MPK}_1 = \ldots = \mathsf{MPK}_T = \mathsf{MPK}$ (the reason for duplicating MPK will be clearer later).

Figure 6: Initialization Circuit C_{INIT}.

- **Sending chains and commitments.** Then S sends R all garbled chains together with a commitment for every label associated with its input wires in all copies $i \in [s]$.

- **Reveal the cut-and-choose parameter.** The receiver R then notifies S with its choice of Z and proves that indeed that is the subset it used in Π_{SCCOT} (by sending a pair of labels for some of its input wires in every chain with an index in Z).

- **Checking correctness of check-chains.** When convinced, S sends R a pair of labels for each input wire associated with the sender's input; this allows R check all the check chains, such that if all found to be built correctly than the majority of the other, evaluation chains, are also built correctly with overwhelming probability.

- **Input consistency.** S then supplies R with a single label for each input wire associated with the sender's input, for all evaluation chains; this step requires checking that those labels are consistent

with a *single* input x_2 of the sender. To this end, S and R run the input consistency protocol that is provided by the garbling scheme defined in 2.6.

- **Evaluation.** Upon verifying their consistency, R uses the input labels and evaluates all evaluation chains, such that in every time step t it discards the chains that their outputs $(i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, \mathsf{sk}_t, \mathsf{translate}_t)$ do not comply to the majority of the outputs in all evaluation chains. We put a spotlight on the importance of the random strings R_1, R_2 that the parties provide to the chains, these allow our protocol to use a *single* block of data D for *all* threads of evaluation, which could not be done in a trivial plugging of the cut-and-choose technique. As explained in Definition 2.8, verifying the correctness of the check chains can be done given only (both of the) input labels for $\mathrm{C}_{\mathrm{INIT}}$ circuits.

Achieving to full security. In the next step we apply the general transformation discussed in Section 2.3, from UMA to full security.

4.1 2PC in the UMA Model
We proceed with the formal detailed description of our protocol.

Protocol Π_{UMA}^{P} executed between sender S and receiver R. Unless stated differently, in the following parameters z, i, t, j respectively iterate over $[Z], [s], [T], [\ell]$.

Inputs. S has input x_1 and R has input x_2 where $|x_1| = |x_2| = \ell'$. R has a blank storage device D with a capacity of n bits.

Auxiliary inputs.

- Security parameters κ and s.

- The description of a program P and a set of circuits $\mathrm{C}_{\mathrm{INIT}}, \mathrm{C}_{\mathrm{CPU+}}^{1}, \ldots, \mathrm{C}_{\mathrm{CPU+}}^{T}$ (as described above) that computes its CPU-steps, such that the output of the last circuit state_{T+1} equals $P^D(x_1, x_2)$, given that the read/write instructions output by the circuits are being followed.

- $(\mathbb{G}, g, q)$ where $\mathbb{G}$ is cyclic group with generator g and prime order q, where q is of length κ.

- S and R respectively choose random strings R_1 and R_2 where $|R_1| = |R_2| = (4t+1) \cdot m$. We denote the overall input size of the parties by ℓ, that is, $|x_1| + |R_1| = |x_2| + |R_2| = \ell' + (4t+1) \cdot m = \ell$. Also, denote the output size by v_{out}.

The protocol.

1. GARBLED CPU-STEP AND INITIALIZATION CIRCUITS.

   (a) Garble the last CPU-step circuit ($t = T$):
      - Choose random labels for the labels corresponding to state_{T+1}.
      - Garble $\mathrm{C}_{\mathrm{CPU+}}^{t}$ by calling

      $$\left(\{\mathrm{GC}_{\mathrm{CPU}}^{t,i}\}_i, \{\mathsf{lbl}_{\mathrm{in},b}^{u,i,t}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^{\kappa}, s, \mathrm{C}_{\mathrm{CPU+}}^{t}, \{\mathsf{lbl}_{\mathrm{out},b}^{v,i,t}\}_{v,i,b}; r_g^t\right)$$

      for $v \in [v_{\mathrm{out}}], i \in [s], b \in \{0,1\}$ and r_g^t the randomness used within Garb.

- Interpret the result labels $\{\mathsf{lbl}_{\mathrm{in},b}^{u,i,t}\}_{u,i,b}$ as the following groups of values: state_t, b_t^{read}, MPK_t, TSK_t and r_t, that cover the labels: $\{\mathsf{lbl}_{\mathsf{state},b}^{u,i,t}\}_{u,i,b}$, $\{\mathsf{lbl}_{b_t^{\mathrm{read}},b}^{u,i,t}\}_{u,i,b}$, $\{\mathsf{lbl}_{\mathsf{MPK}_t,b}^{u,i,t}\}_{u,i,b}$, $\{\mathsf{lbl}_{\mathsf{TSK}_t,b}^{u,i,t}\}_{u,i,b}$, $\{\mathsf{lbl}_{r_t,b}^{u,i,t}\}_{u,i,b}$ resp.

(b) Garble the remaining CPU-step circuits. For $t = T - 1, \ldots, 1$:

- Hard-code the labels $\{\mathsf{lbl}_{b_{t+1}^{\mathrm{read}},b}^{u,i}\}_{u,i,b}$ inside $\mathrm{C}_{\mathrm{CPU+}}^t$.

- Choose random labels for the output wires that correspond to i_t^{read}, i_t^{write}, $\mathsf{sk}_{t,i,b}$ and $\mathsf{translate}_t$ and unite them with the labels $\{\mathsf{lbl}_{\mathsf{state},b}^{u,i,t+1}\}_{u,i,b}$ correspond to state_{t+1} obtained from the previous invocation of Garb; denote the resulting set $\{\mathsf{lbl}_{\mathrm{out},b}^{v,i,t}\}_{v,i,b}$.

- Garble $\mathrm{C}_{\mathrm{CPU+}}^t$ by calling

$$\left(\{\mathrm{GC}_{\mathrm{CPU}}^{t,i}\}_i, \{\mathsf{lbl}_{\mathrm{in},b}^{u,i,t}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, \mathrm{C}_{\mathrm{CPU+}}^t, \{\mathsf{lbl}_{\mathrm{out},b}^{v,i,t}\}_{v,i,b}; r_g^t\right)$$

with $\{\mathsf{lbl}_{\mathrm{out},b}^{v,i,t}\}_{v,i,b}$ the set of labels from above and r_g^t the randomness used within Garb.

- Interpret the result labels $\{\mathsf{lbl}_{\mathrm{in},b}^{u,i,t}\}_{u,i,b}$ as the following groups of values: state_t, b_t^{read}, MPK_t, TSK_t and r_t, that cover the labels: $\{\mathsf{lbl}_{\mathsf{state},b}^{u,i,t}\}_{u,i,b}$, $\{\mathsf{lbl}_{b_t^{\mathrm{read}},b}^{u,i,t}\}_{u,i,b}$, $\{\mathsf{lbl}_{\mathsf{MPK}_t,b}^{u,i,t}\}_{u,i,b}$, $\{\mathsf{lbl}_{\mathsf{TSK}_t,b}^{u,i,t}\}_{u,i,b}$, $\{\mathsf{lbl}_{r_t,b}^{u,i,t}\}_{u,i,b}$ resp.

(c) Garble the initialization circuit $\mathrm{C}_{\mathrm{INIT}}$:

- Combine the group of labels $\{\mathsf{lbl}_{\mathsf{state},b}^{u,i,1}\}_{i,b}$, that is covered by the value state_1 which resulted from the last invocation of Garb, with the groups of labels $\{\mathsf{lbl}_{\mathsf{MPK}_t,b}^{u,i,t}, \mathsf{lbl}_{\mathsf{TSK}_t,b}^{u,i,t}, \mathsf{lbl}_{r_t,b}^{u,i,t}\}_{u,i,b}$ that are covered by the values $\{\mathsf{MPK}_t, \mathsf{TSK}_t, r_t\}$ for all $t \in [T]$. That is, set $\{\mathsf{lbl}_{\mathrm{out},b}^{v,i}\}_{v,i,b} = \{\mathsf{lbl}_{\mathsf{state},b}^{u,i,1} \cup \mathsf{lbl}_{\mathsf{MPK}_t,b}^{u,i,t} \cup \mathsf{lbl}_{\mathsf{TSK}_t,b}^{u,i,t} \cup \mathsf{lbl}_{r_t,b}^{u,i,t}\}_{u,i,b}$ for all u, i, t, b.

- Garble the initialization circuit:

$$\left(\{\mathrm{GC}_{\mathrm{INIT}}^i\}_i, \{\mathsf{lbl}_{\mathrm{in},b}^{u,i}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, \mathrm{C}_{\mathrm{INIT}}, \{\mathsf{lbl}_{\mathrm{out},b}^{v,i}\}_{v,i,b}; r_g^0\right).$$

- Interpret the input labels result from that invocation of Garb by $\{\mathsf{lbl}_{\mathrm{S},b}^{u,i}\}_{u,i,b}$ and $\{\mathsf{lbl}_{\mathrm{R},b}^{u,i}\}_{u,i,b}$ which are the input wire labels that are respectively associated with the sender's and receiver's input wires.

2. Oblivious transfers.

S and R run the Batch Single-Choice Cut-And-Choose Oblivious Transfer protocol Π_{SCCOT}.

(a) S defines vectors $\mathbf{v_1}, \ldots, \mathbf{v_\ell}$ so that $\mathbf{v_j}$ contains the s pairs of random labels associated with R's jth input bit $x_2[j]$ in all garbled circuits $\mathrm{GC}_{\mathrm{INIT}}^1, \ldots, \mathrm{GC}_{\mathrm{INIT}}^s$.

(b) R inputs a random subset $Z \subset [s]$ of size exactly $s/2$ and bits $x_2[1], \ldots, x_2[\ell]$.

(c) The result of Π_{SCCOT} is that R receives *all* the labels associated with its input wires in all circuits $\mathrm{GC}_{\mathrm{INIT}}^z$ for $z \in Z$, and receives a single label for every wire associated with its input x_2 in all other circuits $\mathrm{GC}_{\mathrm{INIT}}^z$ for $z \notin Z$.

3. SEND GARBLED CIRCUITS AND COMMITMENTS.

   S sends R the garbled circuits chains $GC_{INIT}^i, GC_{CPU+}^{1,i}, \ldots, GC_{CPU+}^{T,i}$ for every $i \in [s]$, and the commitment $com_b^{u,i} = Com(lbl_{S,b}^{u,i}, dec_b^{u,i})$ for every label in $\{lbl_{S,b}^{u,i}\}_{u,i,b}$ where $lbl_{S,b}^{u,i}$ is the bth label ($b \in \{0,1\}$) for the sender's uth bit ($u \in [\ell]$) for the ith garbled circuit GC_{INIT}.

4. SEND CUT-AND-CHOOSE CHALLENGE.

   R sends S the set Z along with the *pair* of labels associated with its first input bit in every circuit GC_{INIT}^z for every $z \in Z$. If the values received by S are incorrect, it outputs $\perp$ and aborts. Chains $GC_{INIT}^z, GC_{CPU+}^{1,z}, \ldots, GC_{CPU+}^{t,z}$ for $z \in Z$ are called *check-circuits*, and for $z \notin Z$ are called *evaluation-circuits*.

5. SEND ALL INPUT GARBLED VALUES IN CHECK CIRCUITS.

   S sends the pair of labels and decommitments that correspond to its input wires for every $z \in Z$, whereas R checks that these are consistent with the commitments received in Step 3. If not R aborts, outputting $\perp$.

6. CORRECTNESS OF CHECK CIRCUITS.

   For every $z \in Z$, R has a pair of labels for every input wire for the circuits GC_{INIT}^z (from Steps 2 and 5). This means that it can check the correctness of the chains $GC_{INIT}^z, GC_{CPU+}^{1,z}, \ldots, GC_{CPU+}^{T,z}$ for every $z \in Z$. If the chain was not built correctly for some z then output $\perp$.

7. CHECK GARBLED INPUTS CONSISTENCY FOR THE EVALUATION-CIRCUITS.

   - S sends the labels $\left\{ (lbl_{in,x_1[1]}^{1,z}, \ldots, lbl_{in,x_1[\ell]}^{\ell,z}) \right\}_{z \notin [Z]}$ for its input x_1.
   - S and R participate in the input consistency check protocol Π_{IC}.
     - The common inputs for this protocol are the circuit C_{INIT}, its garbled versions $\{GC_{INIT}^i\}_{z \notin Z}$ and the labels $\left\{ (lbl_{in,x_1[1]}^{1,z}, \ldots, lbl_{in,x_1[\ell]}^{\ell,z}) \right\}_{z \notin [Z]}$ that were sent before.
     - S inputs its randomness r_g^0 and the set of output labels $\{lbl_{out,b}^{v,i}\}_{v,i,b}$ that were used within Garb on input GC_{INIT}, along with the decommitments $\{dec_b^{u,z}\}_{u \in [\ell], z \notin Z, b \in \{0,1\}}$.

8. EVALUATION.

   Let $\tilde{Z} = \{z \mid z \notin Z\}$ be the indices of the *evaluation* circuits.

   (a) For every $z \in \tilde{Z}$, R evaluate GC_{INIT}^z using Eval and the input wires it obtained in Step 7 and reveal one label for each of its output wires $lbl_{INIT}^{out,z}$.

   (b) For $t = 1$ to T:

       i. For every $z \in \tilde{Z}$, evaluate $GC_{CPU+}^{t,z}$ using Eval and obtain one output label for each of its output wires, namely, $lbl_{CPU+}^{out,t,z}$. Part of these labels refer to $state_{t+1,z}$. In addition Eval outputs $out_{t,z} = (i_{t,z}^{read}, i_{t,z}^{write}, b_{t,z}^{write}, translate_{t,z})$ in the clear[12]. For $t = T$ Eval outputs $state_{T+1}$ in the clear and we assign $out_{t,z} = state_{T+1,z}$.

       ii. Take the majority $out_t = Maj(\{out_{t,z}\}_{z \in \tilde{Z}})$ and remove from $\tilde{Z}$ the indices $\tilde{z}$ for which $out_{t,\tilde{z}} \neq out_t$. Formally set $\tilde{Z} = \tilde{Z} \setminus \{z' \mid out_{t,z'} \neq out_t\}$. This means that R halts the execution thread of the circuit copies that were found flawed during the evaluation.

---

[12] Note that if S is honest then $out_{t,z_1} = out_{t,z_2}$ for every $z_1, z_2 \in \tilde{Z}$.

iii. Output out_{T+1}.

The composition theorem of [Can00] allows us to work in a simpler environment where a trusted party is exist for some black-box use of other secure protocol, and so, and we prove the following theorem in the hybrid model (for further details see A.2).

Theorem 4.1. *Assume that π_{GC} is a garbling scheme (cf. Definition 2.8), that π_{TIBE} is TIBE scheme (cf Definition 2.7) and that* Com *is a statistical binding commitment scheme (cf. Definition A.1). Then, protocol Π^P_{UMA} securely realizes $\mathcal{F}_{\mathrm{UMA}}$ in the presence of malicious adversaries in the $\{\mathcal{F}_{\mathrm{SCCOT}}, \mathcal{F}_{\mathrm{IC}}\}$-hybrid for all program executions with ptWrites.*

High-level overview of our proof. In this section we present the intuition of why does our protocol secure in the UMA model, while a full proof of lemma 4.1 presented at Section 5. With respect to garbled circuits security, we stress that neither the selective-bit-attack nor the incorrect-circuit-construction attack can harm the computation here due to the cut-and-choose technique, which prevents the sender from cheating in more than $\frac{s-|Z|}{2}$ of the circuits without being detected. As explained in [LP11], the selective-bit attack cannot be carried out successfully since R obtains all the input keys associated with its input in the cut-and-choose oblivious transfer, where the labels associated with both the check and evaluation circuits are obtained together. Thus, if S attempts to run a similar attack for a small number of circuits then it will not effect the majority, whereas if it does so for a large number of circuits then it will be caught with overwhelming probability. In the protocol, R checks that half of the chains and their corresponding input garbled values were correctly generated. It is therefore assured that with high probability the majority of the remaining circuits and their input garbled values are correct as well. Consequently, the result output by the majority of the remaining circuits must be correct.

The proof for the case the receiver is corrupted is based on two secure components: The garbling scheme and the timed IBE scheme, in the proof we reduce the security of our protocol to the security of each one of them. The intuition behind this proof asserts that R receives $|Z|$ opened check circuits and $|\tilde{Z}| = s - |Z|$ evaluation circuits. Such that for each evaluation circuit it only receives a single set of keys for decrypting the circuit. Furthermore, the keys that it receives for each of the $|\tilde{Z}|$ evaluation circuits are associated with the same pair of inputs x_1, x_2. This intuitively implies that R can do nothing but correctly decrypt $|\tilde{Z}|$ circuits, obtaining the same value $P^d(x_1, x_2)$. One concern regarding the security proof stems from the use of a TIBE encryption scheme within each of the CPU-step circuits. Consequently, we have to argue the secrecy of the input label that is not decrypted by R. Specifically, we show that this is indeed the case by constructing a simulator that, for each CPU-step, outputs a fake translate table translate that correctly encrypts the active label (namely, the label observed by the adversary), yet encrypts a fake inactive label. We then show, that the real view in which all labels are correctly encrypted, is indistinguishable from the simulated view in which only the active label is encrypted correctly.

5 Proof of Theorem 4.1

We prove Theorem 4.1 in a hybrid model where a trusted party is used to compute the batch single-choice cut-and-choose oblivious transfer functionality $\mathcal{F}_{\mathrm{CT}}$ and the input consistency check functionality $\mathcal{F}_{\mathrm{IC}}$. We separately prove the case that S is corrupted and the case that R is corrupted.

5.1 The Case S is Corrupted

This case is very similar to the case in which S is corrupted in a standard cut-and-choose based protocol (e.g. [LP11]). Intuitively, S can only cheat by constructing some of the circuits in an incorrect way. However, in order for this to influence the outcome of the computation, it has to be that a majority of the evaluation circuits, or equivalently over one quarter of them, are incorrect. Furthermore, it must hold that none of these incorrect circuits are part of the check circuits. The reason this bad event only occurs with negligible probability is that S is committed to the circuits *before* it learns which circuits are the check circuits and which are the evaluation circuits. Specifically, observe first that in protocol Π_{SCCOT}, R receives all the keys associated with its own input wires for the check circuits in Z (while S knows nothing about Z). Furthermore, S sends commitments for all input wire labels for input wires associated with its input before learning Z. Thus, it can only succeed in cheating if it successfully guesses over $s/4$ circuits which all happen to not be in Z. As shown in [LP11], this event occurs with probability of approximately $\frac{1}{2^{s/4}}$. The sender S further participates in an input consistency protocol Π_{IC} which proves to R that all its inputs to the evaluation circuits are consistent.

We now proceed to the formal proof. Let $\mathcal{A}$ be an adversary controlling S in an execution of Π^P_{UMA} where a trusted party is used to compute the cut-and-choose OT functionality $\mathcal{F}_{\text{SCCOT}}$ and the input consistency check functionality $\mathcal{F}_{\text{IC}}$. We construct a simulator $\mathcal{S}$ that runs in the ideal model with a trusted party computing $\mathcal{F}^{PD}_{\text{UMA}}$. The simulator $\mathcal{S}$ internally invokes $\mathcal{A}$ and simulates the honest R for $\mathcal{A}$ as well as the trusted party computing $\mathcal{F}_{\text{SCCOT}}$ and $\mathcal{F}_{\text{IC}}$ functionalities. In addition, $\mathcal{S}$ interacts externally with the trusted party computing $\mathcal{F}^{PD}_{\text{UMA}}$. $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ upon its input and receives the inputs that $\mathcal{A}$ sends to the trusted party computing $\mathcal{F}_{\text{SCCOT}}$ functionality. These inputs constitute an $\ell \times s$ matrix of label pairs $\{(\text{lbl}^{1,1}_{\text{R},0}, \text{lbl}^{1,1}_{\text{R},1}), \ldots, (\text{lbl}^{\ell,s}_{\text{R},0}, \text{lbl}^{\ell,s}_{\text{R},1})\}$, where $\text{lbl}^{j,i}_{\text{R},b}$ is the label associated with the jth input wire of the receiver R in the ith garbled version of the circuit C_{INIT}. Recall that these labels constitute the garbled x_1 and R_1 for all chains $i \in [s]$.

2. $\mathcal{S}$ receives from $\mathcal{A}$ s garbled chains $\text{GC}^i_{\text{INIT}}, \text{GC}^{1,i}_{\text{CPU+}}, \ldots, \text{GC}^{T,i}_{\text{CPU+}}$ for every $i \in [s]$ and $2\ell s$ commitments $\{\text{com}^{u,i}_b\}$ for every label $\text{lbl}^{u,i}_{\text{S},b}$ as described in the protocol (the garbled values associated with the sender's input wires to $\{\text{C}_{\text{INIT}}\}_i$ for all $i \in [s]$).

3. $\mathcal{S}$ chooses a subset $Z \subset [s]$ of size $s/2$ uniformly at random. For every $z \in Z$, $\mathcal{S}$ hands $\mathcal{A}$ the values $\text{lbl}^{1,z}_{\text{R},0}, \text{lbl}^{1,z}_{\text{R},1}$ (i.e. the two labels for the first input wire of R in every check chain, this proves to $\mathcal{A}$ that this chain is indeed a check chain, otherwise, R could not know *both* of the labels for that wire).

4. $\mathcal{A}$ sends the decommitments to all labels of its input wires for the check chains (i.e. all chains indexed by $z \in Z$). Namely, upon receiving the set $\{\text{lbl}^{u,i}_{\text{S},b}, \text{dec}^{u,i}\}$ where $\text{lbl}^{u,i}_{\text{S},b}$ is the bth label ($b \in \{0,1\}$) for the sender's uth bit ($u \in [\ell]$) for the ith garbled circuit GC_{INIT} and $\text{dec}^{u,i}_b$ is its decommitment information. $\mathcal{S}$ verifies that the decommitment information is correct. If not, $\mathcal{S}$ sends $\bot$ to the trusted party, simulates R aborting and outputs whatever $\mathcal{A}$ outputs.

5. $\mathcal{S}$ verifies that all the check chains $\text{GC}^z_{\text{INIT}}, \text{GC}^{1,z}_{\text{CPU}}, \ldots, \text{GC}^{T,z}_{\text{CPU}}$ for $z \in Z$ are correctly constructed (the same way that an honest R would). If not, it sends $\bot$ to the trusted party, simulates R aborting and outputs whatever $\mathcal{A}$ outputs.

6. $\mathcal{S}$ receives labels $\left\{ (\hat{\text{lbl}}^{1,z}_{\text{in},x_1[1]}, \ldots, \hat{\text{lbl}}^{\ell,z}_{\text{in},x_1[\ell]}) \right\}_{z \notin [Z]}$. In addition $\mathcal{S}$, as a trusted party in the input consistency protocol Π_{IC}, receives the randomness r^0_g, the output labels $\{\text{lbl}^{v,i}_{\text{out},b}\}_{v,i,b}$ that were used by $\mathcal{A}$

27

to generate the s garbled chains in Step 1 of the protocol, together with the decommitments $\mathsf{dec}_b^{u,i}$ for every label associated with the sender's input wires.

7. Given the values in the previous step, $\mathcal{S}$ checks the consistency of the labels it received from S (as if the trusted party in $\mathcal{F}_{\mathrm{IC}}$ would). Note that if the check follows, the simulator $\mathcal{S}$ is able to extract the sender's input x_1.

   - If $\mathcal{F}_{\mathrm{IC}}$ returns 0 then $\mathcal{S}$ outputs $\bot$, simulates R aborting and outputs whatever $\mathcal{A}$ outputs.
   - Otherwise, for every $u \in [\ell']$ ($|x_1| = \ell'$ as specified above), if $\hat{\mathsf{lbl}}_{\mathrm{in},x_1[u]}^{u,z} = \mathsf{lbl}_{\mathrm{in},0}^{u,z}$ set $x_1[u] = 0$ and if $\hat{\mathsf{lbl}}_{\mathrm{in},x_1[u]}^{u,z} = \mathsf{lbl}_{\mathrm{in},1}^{u,z}$ set $x_1[u] = 1$. Note that $\mathcal{S}$ only extracts the values associated with x_1 and not R_1.

8. $\mathcal{S}$ sends $(\mathrm{INPUT_S}, x_1)$ to the trusted party computing $\mathcal{F}_{\mathrm{UMA}}^{P^D}$ and outputs whatever $\mathcal{A}$ outputs and halts.

We next prove that for every $\mathcal{A}$ corrupting S and every s it holds that

$$\left\{ \mathbf{IDEAL}_{\mathcal{S}(z),\mathrm{S}}^{\mathcal{F}_{\mathrm{UMA}}^{P^D}}(\kappa, x_1, x_2) \right\}_{\kappa \in \mathbb{N}, x_1, x_2, z \in \{0,1\}^*} \overset{\kappa,s}{\approx} \left\{ \mathbf{REAL}_{\mathcal{A}(z),\mathrm{S}}^{\Pi_{\mathrm{UMA}}^P}(\kappa, x_1, x_2) \right\}_{\kappa \in \mathbb{N}, x_1, x_2, z \in \{0,1\}^*}$$

The sender's view in our protocol is very limited, the values that it sees during the execution are (1) the set of indices Z (in Step 4 of the protocol), (2) $|Z|$ pairs of labels that proves that that Z is indeed the one used in Π_{SCCOT}, and (3) the output of the execution. The simulator $\mathcal{S}$ chooses Z exactly as a honest receiver would and sends S the rest of the values correctly (also, if it caught a cheat, it aborts as a honest receiver would do). Importantly, is we argue immediately, the adversary could not deviate from the protocol (and produce sufficient amount of incorrect garbled chains) without being caught with overwhelming probability.

Note that after Step 3 all labels for input wires of S, and all garbled chains are *fully determined*, also, one label for every input wire associated with R is fully determined as well. Therefore, after this step each of the chain of circuits $\mathrm{GC}_{\mathrm{INIT}}^i, \mathrm{GC}_{\mathrm{CPU+}}^{1,i}, \ldots, \mathrm{GC}_{\mathrm{CPU+}}^{T,i}$ is either "bad" or "not bad".

It was previously shown, with regard to cut-and-choose analysis, that the probability that R does not abort and yet the majority of the *evaluation* circuits is bad, is at most $\frac{1}{2^{s/4}}$. We denote this above event by $\mathsf{badMaj} \wedge \mathsf{noAbort}$ and claim that as long that this event does not occur, the result of the ideal and hybrid executions (where the oblivious transfers and input consistency are ideal) are identically distributed. This is due to the fact that if less than $s/4$ circuits are bad, then the majority of circuits evaluated by R compute the correct chain of circuits $\mathrm{GC}_{\mathrm{INIT}}^i, \mathrm{GC}_{\mathrm{CPU+}}^{1,i}, \ldots, \mathrm{GC}_{\mathrm{CPU+}}^{T,i}$ which in turn correctly evaluates the program P^D due to the correctness of the garbled scheme. In addition, by the ideal input consistency, the input x_1 extracted by the simulator $\mathcal{S}$ and sent to the trusted party computing P^D corresponds exactly to the input x_1 in the computation of every not-bad chain of circuits. Thus, in every not-bad chain R outputs $P^D(x_1, x_2)$, and this is the majority of the evaluation circuits. We conclude that as long as $\mathsf{badMaj} \wedge \mathsf{noAbort}$ does not occur, R outputs $P^D(x_1, x_2)$ in both the real and ideal executions. Finally, we observe that $\mathcal{S}$ sends $\bot$ to the trusted party whenever R would abort and output $\bot$. This completes the proof of this corruption case.

5.2   The Case R is Corrupted

The intuition of this proof is as follows. For each of the evaluation chains the receiver only receives a single set of input labels. Furthermore, these labels are associated with the same pair of inputs x_1, x_2 due to the single-choice cut-and-choose OT $\mathcal{F}_{\mathrm{SCCOT}}$ functionality. This implies that R can do nothing but honestly evaluating the evaluation circuits, where each final circuit outputs the same value $P^D(x_1, x_2)$. That it,

assume that R evaluates the $s/2$ garbled circuits $\text{GC}_{\text{CPU}^+}^{t,z}$ for CPU-step t and all $z \notin Z$; these garbled circuits output a translate translate_t tuple which corresponds to the values $\text{ct}_0 = \text{Enc}_{\text{MPK}}(\text{id}_0, \text{lbl}_0^{t+1})$ and $\text{ct}_1 = \text{Enc}_{\text{MPK}}(\text{id}_1, \text{lbl}_1^{t+1})$.[13] Now, since R only knows a secret key for the identity id_b from a previous write operation, yet it does not know the secret key that is associated with id_{1-b} it can only decrypt ct_b. Below we formalize this intuition, namely, we show that R cannot learn any significant information about the plaintext within ct_{1-b} and thus, cannot extract the other label lbl_{1-b} for the next CPU step circuit.

Let $\mathcal{A}$ be an adversary controlling R in an execution of protocol Π_{UMA}^P where a trusted party is used to compute the cut-and-choose OT functionality $\mathcal{F}_{\text{SCCOT}}$ and the input consistency functionality $\mathcal{F}_{\text{IC}}$. We construct a simulator $\mathcal{S}$ for the ideal model with a trusted party computing $\mathcal{F}_{\text{UMA}}^{PD}$.

1. $\mathcal{S}$ invokes $\mathcal{A}$ upon its input and receives its inputs to the trusted party computing $\mathcal{F}_{\text{SCCOT}}$. These inputs consist of a subset $Z \subset [s]$ of size exactly $s/2$ and bits $x_2[1], \ldots, x_2[\ell]$. (If Z is not of size exactly $s/2$ then $\mathcal{S}$ simulates S aborting, sends $\perp$ to the trusted party computing $\mathcal{F}_{\text{UMA}}^{PD}$, and halts outputting whatever A outputs.)

2. $\mathcal{S}$ sends (INPUT_R, x_2) to the trusted party computing $\mathcal{F}_{\text{UMA}}^{PD}$ and receives the output (OUTPUT_R, T, y) and the memory accesses $\text{MemAccess} = \{(i_t^{\text{read}}, i_t^{\text{write}}, b_t^{\text{write}})\}_{t \in [T]}$ where i_t^{read} is the address to read from, i_t^{write} is the address to write to and b_t^{write} is the bit value to be written to i_t^{write} in time step t.

3. $\mathcal{S}$ builds s chains of garbled circuits, starting from the last CPU step T towards the first one, in the following manner. (Note that a single call to SimGC produces both the evaluation and the check circuits).

    (a) Initialize the TIBE scheme: generate the keys $(\text{MPK}, \text{MSK}) \leftarrow \text{MasterGen}(1^\kappa)$ and $\text{TSK}_t \leftarrow \text{TimeGen}(\text{MSK}, t)$ for $t = 1, \ldots, T$.

    (b) For the last time step $t = T$, create $\{\text{GC}_{\text{CPU}^+}^{t,z}\}_z$ by calling SimGC on the circuit $\text{C}_{\text{CPU}^+}^t$ such that for the evaluation circuits ($z \notin Z$) the output labels state_{t+1} are set to the value y in the clear, whereas for the check circuits ($z \in Z$) the simulator chooses random pairs of output labels. This produces the input labels for the input state_t and the bit b_t^{read}.

    (c) For any other $t = T - 1 \ldots 1$, recall first that the values $i_t^{\text{read}}, i_t^{\text{write}}, b_t^{\text{write}}$ are given in the clear (from MemAccess). Also, note that the labels $\text{lbl}_{\text{read},b}^{t+1,z}$ for the input bit b of circuit $\text{GC}_{\text{CPU}^+}^{t+1,z}$ had been produced by the simulator in step $t + 1$. The simulator $\mathcal{S}$ computes the secret key $\text{sk}_{(t,i,b)}$ and the translation table translate_t as follows:

      - Let $i = i_t^{\text{write}}$ and $b = b_t^{\text{write}}$. If $i = \perp$ then set $\text{sk}_{(t,i,b)} := \perp$. Else, set $\text{sk}_{(t,i,b)} \leftarrow \text{KeyGen}(\text{TSK}_t, \text{id} = (t, i, b))$.
      - Let $i = i_t^{\text{read}}$, $t' < t$ be the last write-time to location i (i.e., the largest value such that $i_{t'}^{\text{write}} = i_t^{\text{read}}$) and let $b = b_{t'}^{\text{write}}$ be the bit written to the location at time t' (this can be easily computed given MemAccess). Then, set:

$$\text{ct}_b \leftarrow \text{Enc}_{\text{MPK}}((t', i, b), \text{lbl}_{\text{read},b}^{t+1,z}), \quad \text{ct}_{1-b} \leftarrow \text{Enc}_{\text{MPK}}((t', i, b), 0)$$

      for all $z \notin Z$, and set $\text{translate}_t = (\text{ct}_0, \text{ct}_1)$.

---

[13]Recall that all the circuits evaluated in time t output the same value for translate_t since they all use the same randomness to compute it.

(d) Generate $\{GC_{CPU+}^{t,z}\}_z$ by calling SimGC on the circuit C_{CPU+}^t such that for the evaluation circuits ($z \notin Z$) it inputs the values $i_t^{\text{write}}, i_t^{\text{read}}, \text{sk}_{(t,i,b)}, \text{translate}_t$ as output labels and for the check circuits ($z \in Z$) it inputs random pairs of labels. Note that when $t = 1$, the input labels produced by SimGC for state_1 actually refer to the parties inputs x_1, x_2.

(e) At this point, the input labels for all CPU-step circuits $\{GC_{CPU+}^{1,z}, \ldots, GC_{CPU+}^{T,z}\}_z$ are known to $\mathcal{S}$ (specifically, these correspond to either a single label per wire for $z \notin Z$, or a pair of labels per wire for $z \in Z$). These constitute the output labels that are required for SimGC to simulate the initialization circuits $\{GC_{INIT}^z\}_z$. Namely, we have the output labels for x_1, x_2 and $\{\text{MPK}_t, \text{TSK}_t, r_t^{\text{KeyGen}}, r_{t,0}^{\text{Enc}}, r_{t,1}^{\text{Enc}}\}_{t \in [T]}$ (again, a single label if $z \notin Z$ and pair of labels if $z \in Z$). The simulator $\mathcal{S}$ inputs these labels as the output labels to SimGC which produces the labels for the input wires of the circuits $\{C_{INIT}^z\}_z$.

4. Let $\tilde{Z} = s \setminus Z$ be the indices of the evaluation chains. Then in the previous step the simulator produced s sets of labels. For chains indexed with $z \in Z$ (*check* chain) the set consists of ℓ *pairs* of labels corresponding to R's inputs wires in GC_{INIT}^z, whereas for chains indexed with $z \in \tilde{Z}$ (evaluation chains) the set consists of ℓ *single* labels corresponding to R actual input x_2. These $(2\ell|Z| + \ell|\tilde{Z}|)$ labels are denoted by $\overline{\text{lbl}_Z} = (\text{lbl}_{R,0}^{1,z}, \text{lbl}_{R,1}^{1,z}, \ldots, \text{lbl}_{R,0}^{\ell,z}, \text{lbl}_{R,1}^{\ell,z})$ for all $z \in Z$, and by $\overline{\text{lbl}_{\tilde{Z}}} = (\text{lbl}_{R,x_2[1]}^{1,z}, \ldots, \text{lbl}_{R,x_2[\ell]}^{\ell,z})$ for $z \in \tilde{Z}$. Then, $\mathcal{S}$ hands $\mathcal{A}$ all the above labels, i.e. the union $\overline{\text{lbl}_Z} \cup \overline{\text{lbl}_{\tilde{Z}}}$ as its output from the oblivious transfers. (Note that $\mathcal{S}$ knows x_2 because it extracted it in the beginning of the simulation).

5. The simulator $\mathcal{S}$ sends $\mathcal{A}$ the garbled chains and commitments on the labels of all input wires of circuits $\{GC_{INIT}^i\}_{i \in [s]}$.

6. $\mathcal{S}$ receives the set Z' along with a pair of labels for every $z \in Z$ (proving that $\mathcal{A}$ indeed entered Z).

   (a) If $Z \neq Z'$ and yet the values received are all correct then $\mathcal{S}$ outputs $\bot$ and halts.

   (b) If $Z = Z'$ and any of the values received are incorrect, then $\mathcal{S}$ sends $\bot$ to the trusted party, simulates S aborting. and halts outputting whatever $\mathcal{A}$ outputs.

   (c) Otherwise, $\mathcal{S}$ proceeds as below.

7. $\mathcal{S}$ hands $\mathcal{A}$ the input labels that correspond to the sender's input for all $z \notin Z$ and $u \in [\ell]$, and sends the value 1 as the output of the trusted party when using the input consistency check functionality $\mathcal{F}_{\text{IC}}$.

8. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

We now show that for every $\mathcal{A}$ corrupting R and every s it holds that:

$$\left\{ \textbf{IDEAL}_{\mathcal{S}(z),R}^{\mathcal{F}_{UMA}^{P^D}}(\kappa, x_1, x_2) \right\}_{\kappa \in \mathbb{N}, x_1, x_2, z \in \{0,1\}^*} \overset{\kappa,s}{\approx} \left\{ \textbf{REAL}_{\mathcal{A}(z),R}^{\pi}(\kappa, x_1, x_2) \right\}_{\kappa \in \mathbb{N}, x_1, x_2, z \in \{0,1\}^*}$$

In order to do so, we define a series of hybrid distributions $\textbf{Hyb}_t$ for $t = 1, \ldots, T$. In the hybrid t, the garbled CPU step circuits $GC_{CPU+}^{1,z}, \ldots, GC_{CPU+}^{t,z}$ for $z \in Z$ are created as in the simulation and the garbled CPU step circuits $GC_{CPU+}^{t+1,z}, \ldots, GC_{CPU+}^{T,z}$ for $z \in Z$ are created as in the real distribution (that is, both labels $\text{lbl}_{\text{read},0}^{t+1,z}, \text{lbl}_{\text{read},1}^{t+1,z}$ for the input bit of the next circuit are encrypted). In $\textbf{Hyb}_t$, when we simulate the tth circuits $GC_{CPU+}^{t,z}$, we use the output labels for $\text{state}_{t+1}, b_{t+1}^{\text{read}}$ that these wire takes on during the real computation.

We also define a hybrid distribution $\mathbf{Hyb}'_t$ which is like $\mathbf{Hyb}_t$ except for the simulation of the tth CPU step circuits $GC^{t,z}_{\text{CPU}^+}$ for $z \in Z$. Instead of choosing translate_t as in the simulation described above, we choose $\text{translate}_t = (ct_0, ct_1)$ to both be encryptions of the correct label of the next circuit:

$$ct_b \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\text{read}}_t, b), \mathsf{lbl}^{t+1,z}_{\text{read},b}) \quad , \quad ct_b \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\text{read}}_t, b), \mathsf{lbl}^{t+1,z}_{\text{read},1-b})$$

where $\mathsf{lbl}^{t+1,z}_{\text{read},0}), \mathsf{lbl}^{t+1,z}_{\text{read},1}$ are the labels corresponding to the bits $0, 1$ for the wire b^{write}_{t+1} in garbled circuit $GC^{t+1,z}_{\text{CPU}^+}$, which is still created using the real garbling procedure. (If $t = T$ we define $\mathbf{Hyb}'_t$ to be the same as $\mathbf{Hyb}_t$).

Note that in $\mathbf{Hyb}_0$ none of the CPU step circuits are simulated, yet, the initialization circuits GC^z_{INIT} are still simulated. Therefore, we define the hybrid $\mathbf{Hyb}_{(-1)}$ to be the distribution where all circuits are created as in the real distribution.

Note that $\mathbf{Hyb}_{-1}$ is equal to the real distribution and $\mathbf{Hyb}_T$ is equal to the simulated distribution. Therefore, we prove indistinguishability by showing that for each t, we have:

$$\mathbf{Hyb}_t \stackrel{c}{\approx} \mathbf{Hyb}'_{t+1} \stackrel{c}{\approx} \mathbf{Hyb}_{t+1}$$

and

$$\mathbf{Hyb}_{(-1)} \stackrel{c}{\approx} \mathbf{Hyb}_0$$

We prove this by the following claims:

Claim 5.1. *For each $t \in \{0, \ldots, T\}$ it holds that $\mathbf{Hyb}_t \stackrel{c}{\approx} \mathbf{Hyb}'_{t+1}$.*

Proof: This follows directly from the security of the circuit garbling scheme applied only to the garbled CPU set of circuits for step $t + 1$. This is because, in $\mathbf{Hyb}_t$, all of the circuits $GC^{1,z}_{\text{CPU}^+}, \ldots, GC^{t,z}_{\text{CPU}^+}$ are already simulated and hence they only rely on a subset of the input wire labels for the input $\text{state}_{t+1}, b^{\text{write}}_{t+1}$, in the $t+1$th set of circuits, corresponding to the actual values that these wires should take on during the real computation. (this is true for the wire corresponding to b^{write}_{t+1} since the simulated translate_t used to create the tth circuit only encrypts one label and the other ciphertext is "dummy".) ∎

Claim 5.2. *For each $t \in \{0, \ldots, T\}$ it holds that $\mathbf{Hyb}'_t \stackrel{c}{\approx} \mathbf{Hyb}_t$.*

Proof: This follows directly from the security of the TIBE scheme. The only difference between $\mathbf{Hyb}'_t$ and $\mathbf{Hyb}_t$ is the value of $\text{translate}_t = (ct_0, ct_1)$ used to simulate the tth set of circuits. Let $b = b^{\text{write}}_{t+1}$ be the value of the read-bit in location i^{read}_t in the computation. Then, in $\mathbf{Hyb}'_t$ we set

$$ct_b \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\text{read}}_t, b), \mathsf{lbl}^{t+1,z}_{\text{read},b}) \quad , \quad ct_b \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\text{read}}_t, b), \mathsf{lbl}^{t+1,z}_{\text{read},1-b})$$

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

$$ct_b \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\text{read}}_t, b), \mathsf{lbl}^{t+1,z}_{\text{read},b}) \quad , \quad ct_b \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\text{read}}_t, b), 0)$$

where $u < t$.

Therefore we reduce this to the TIBE game where the adversary is given the master public key MPK, the timed-keys $\mathsf{TSK}_{t+1}, \ldots, \mathsf{TSK}_T$, a single identity secret key for the identity $(t', i^{\text{write}}_{t'}, b^{\text{write}}_{t'})$ for each time step $0 < t' < t$ (this key is used to simulate the set of circuits for time step t').

Assume the existence of parties inputs x_1, x_2 for which there exists a distinguisher $\mathcal{D}$ for the hybrids $\mathbf{Hyb}'_t$ and $\mathbf{Hyb}_t$. We construct a distinguisher $\mathcal{D}'$ for the TIBE scheme. $\mathcal{D}'$ is given MPK, $\mathsf{TSK}_{t+1}, \ldots, \mathsf{TSK}_T$ from the game along with one secret key for every time step $t' < t$. $\mathcal{D}'$ works as follows:

1. Build the circuits $\mathrm{GC}_{\mathrm{CPU+}}^{T,z}, \ldots, \mathrm{GC}_{\mathrm{CPU+}}^{t+1,z}$ for all $z \notin Z$ as in the real distribution.

2. For the tth circuits $\mathrm{GC}_{\mathrm{CPU+}}^{t,z}$, let $b = b_t^{\mathsf{read}}$ be the bit that is being read from memory at time t in the real execution of the program ($\mathcal{D}'$ knows it since it knows x_1, x_2 and can infer b from it) and let $\mathsf{lbl}_{\mathsf{read},b}^{t+1,z}, \mathsf{lbl}_{\mathsf{read},1-b}^{t+1,z}$ be the labels of the input bits for the next CPU step circuits ($\mathcal{D}'$ knows them as well because it generated these labels using Garb).

3. $\mathcal{D}'$ hands the TIBE game the identity $\mathsf{id}^* = (t, i_t^{\mathsf{write}}, b)$ and the two messages: $\mathsf{msg}_0 = \mathsf{lbl}_{\mathsf{read},1-b}^{t+1,z}$ and $\mathsf{msg}_0 = 0$ and receives the ciphertext ct.

4. Set $\mathsf{translate}_t = (\mathsf{ct}_0, \mathsf{ct}_1)$ where $\mathsf{ct}_b = \mathsf{Enc}_{\mathsf{MPK}}((t', i_t^{\mathsf{read}}, b), \mathsf{lbl}_{\mathsf{read},b}^{t+1,z})$ where t' the last time that location i_t^{read} was written to, and $\mathsf{ct}_{1-b} = \mathsf{ct}$.

5. For CPU step circuits $\mathrm{GC}_{\mathrm{CPU+}}^{t,z}$, use the suitable input labels state_{t+1} that was output from the previous invocation of Garb, and the values $i_t^{\mathsf{write}}, i_t^{\mathsf{read}}, b_t^{\mathsf{write}}, \mathsf{translate}_t$ that are output "in the clear" and input them to SimGC to get the appropriate input labels for CPU step circuits $\mathrm{GC}_{\mathrm{CPU+}}^{t-1,z}$.

6. Keep the simulation till the $\mathrm{C}_{\mathrm{INIT}}$ and hand the result garbled chains together with the memory accesses to $\mathcal{D}$.

7. If $\mathcal{D}$ outputs $\mathbf{Hyb}_t'$ then output 0, otherwise, if $\mathcal{D}$ outputs $\mathbf{Hyb}_t$ output 1.

Note that if $\mathsf{ct} = \mathsf{Enc}_{\mathsf{MPK}}((t', i_t^{\mathsf{read}}, b), \mathsf{lbl}_{\mathsf{read},b}^{t+1,z})$ then the result hybrid is identically distributed to $\mathbf{Hyb}_t'$ and if $\mathsf{ct} = \mathsf{Enc}_{\mathsf{MPK}}((t', i_t^{\mathsf{read}}, b), 0)$ then the result hybrid is identically distributed to $\mathbf{Hyb}_t$. Thus, if $\mathcal{D}$ distinguishes between the two hybrids $\mathbf{Hyb}_t'$ and $\mathbf{Hyb}_t$ then the distinguisher $\mathcal{D}'$ distinguish between the above messages in the TIBE game. ∎

Claim 5.3. *It holds that* $\mathbf{Hyb}_{(-1)} \overset{\mathrm{c}}{\approx} \mathbf{Hyb}_0$.

Proof: Note that the difference between the hybrids is merely whether the first circuits $\mathrm{C}_{\mathrm{INIT}}^z$ are simulated or not. Hence, we rely on the security of the garbling scheme as done in the proof of Claim 5.2. ∎

6 Removing the IBE Assumption

Next, we demonstrate how to apply our technique to the construction of Garg et al. [GLOS15]. We start be putting the theorem that is being shown in this section and then we describe how the garbled RAM primitive is being achieved by only assuming the existence of one-way-functions (OWF).

Theorem 6.1. *Under the standard assumptions for achieving static malicious 2PC security, there exists a constant round protocol that securely realizes any RAM program in the presence of malicious adversaries, making only black-box use of an Oblivious RAM construction, where the size of the garbled database is* $|D| \cdot \mathsf{poly}(\kappa, \log n)$ *the size of the garbled input is* $|x| \cdot O(\kappa) + T \cdot \mathsf{poly}(\kappa)$ *and the size of the garbled program and its evaluation time is* $|\mathrm{C}_{\mathrm{CPU}}^P| \times T \times \mathsf{poly}(\kappa, \log(n)) \times s$.

6.1 GRAM Based on OWF [GLOS15]

As in the previous GRAM schemes, the garbler first garble the data D, the program P and the input x, and forwards these to the evaluator who runs the evaluation algorithm to obtain the program output y.

6.1.1 Garbling Data

- Let the data D be of n blocks $D_0, D_1, \ldots, D_{n-1}$, each of size κ bits. These blocks will be the leafs of a binary tree of depth $d = \log n$ such that its $n - 1$ internal nodes are determined next.

- Choose a set of $n - 1$ random keys from $\{0, 1\}^\kappa$ indexed by the tuple i, j where i is the depth in the tree and j is the index of that node within the i-th tree level, where that the i-th level includes 2^i keys. For example, when $|D| = 8\kappa$ then $n = 8$ and the nodes in the tree store the values: $r_{0,0}, r_{1,0}, r_{1,1}, r_{2,0}, r_{2,1}, r_{2,2}, r_{2,3}, D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$ where $r_{0,0}$ is the root. This tree is the *plain data*, whereas the *encrypted data* is described next.

- Encrypt each value within a node (expect for the root) using the key that is associated with its parent node, where the encryption is carried out using a PRF F and the tags left and right. For example, the block $r_{2,1}$ is encrypted using the key $r_{1,0}$ (its parent in the tree) with the tag right (since $r_{2,1}$ is the right child of $r_{1,0}$). That is, $\widehat{r_{2,1}} = F_{r_{1,0}}(\mathsf{right}, r_{2,1})$. Similarly, $\tilde{D}_2 = F_{r_{2,1}}(\mathsf{left}, D_2)$.

- For each plain key $r_{i,j}$ output its encrypted version $\widehat{r_{i,j}}$ and for each plain data block D_j output its encrypted version $\tilde{D}_j$. This entire tree of blocks is the garbled data $\tilde{D}$.

6.1.2 Garbling Program

Each step in the original program P is performed using $d = \log n$ steps in the new scheme, such that the first $\log n - 1$ steps are used to navigate to the required data block in D_j, whereas the last step is used to perform the CPU-step of the program on this block and to kick start the next CPU-step. Concretely, the first $\log n - 1$ navigation steps are carried out by evaluating $\log n - 1$ nav garbled circuits and the last ($\log n$-th) step is performed by evaluating a single garbled circuit step. In the following, we further elaborate on the internal design of the nav and step circuits:

The nav circuit. We index each nav circuit by the tuple i, j ($i \in [T]$ and $j \in [d]$) such that circuit $\mathsf{nav}_{i,j}$ is the j-th circuit in the i-th group of circuits. $\mathsf{nav}_{i,j}$ helps to navigate in the j-th level of the tree towards the data item D_L that is required in the i-th CPU-step. The circuit $\mathsf{nav}_{i,j}$ is given as input two *plain* keys $r_\ell, r_{\ell+1}$ ($\ell \in [2^{j+1}]$) from level j along with the current state of the program and the location L required for the next CPU-step within program P. In addition, the values j, s, r' and $\mathsf{label}_0^{\mathsf{left}}, \mathsf{label}_1^{\mathsf{left}}, \mathsf{label}_0^{\mathsf{right}}, \mathsf{label}_1^{\mathsf{right}}$ are hardwired in $\mathsf{nav}_{i,j}$, where s is the key of the parent of the two keys r_ℓ and $r_{\ell+1}$, r' is a fresh key that will replace either r_ℓ or $r_{\ell+1}$ (depends on the location L). Moreover, the set of labels label_b^x are the input labels that correspond to two keys in the next navigation circuit (for $x \in \{\mathsf{left},\mathsf{right}\}$, $b \in \{0, 1\}$). The output of $\mathsf{nav}_{i,j}$ are the result of F applied on the two pair of keys using the parent key s, and a translation table that initiates the evaluation of the next circuit in the i-th group of circuits, which is either circuit $\mathsf{nav}_{i,j+1}$ or step_i. In more details, the $\mathsf{nav}_{i,j}$ circuit works as follows:

  - If $L_j = 0$ (namely, the j-th bit of L), then replace r_ℓ with r', otherwise replace $r_{\ell+1}$ with r' (remember that r_ℓ and $r_{\ell+1}$ are given as inputs). Call the new key pair $\widehat{r_\ell}, \widehat{r_{\ell+1}}$. Encrypt this pair using their parent's key s (hardwired) and output: $\mathsf{write} = \left(F_s(\widehat{r_\ell}), F_s(\widehat{r_{\ell+1}}), L, \mathsf{state}\right)$. The two ciphertexts would be written to the appropriate locations, (i, ℓ) and $(i, \ell + 1)$, in memory while the location L and the state are passed to the next navigation circuit.

33

– Note that if $L_j = 0$ this means that in the next level of the tree (i.e. level $j + 1$) we wish to use the children of the left key r_ℓ, denoted by r_m and r_{m+1}. Whereas, if $L_j = 1$ then we wish to use the children of the right key $r_{\ell+1}$, denoted by r'_m and $r_{m'+1}$.

W.l.o.g assume that $L_j = 0$, then the current navigation circuit $\mathsf{nav}_{i,j}$ outputs a translation table that enables the evaluator to evaluate the next navigation circuit $\mathsf{nav}_{i,j+1}$ which receives two keys from the $j+1$ level of the tree, namely, the keys r_m, r_{m+1}. Thus the current navigation circuit outputs the translation table $\mathsf{translate} = F_{r_\ell}(0, \mathsf{left}) \oplus \mathsf{label}_0^{\mathsf{left}}$, $\mathsf{translate} = F_{r_\ell}(1, \mathsf{left}, 1) \oplus \mathsf{label}_1^{\mathsf{left}}$ and $\mathsf{translate} = F_{r_\ell}(0, \mathsf{right}) \oplus \mathsf{label}_0^{\mathsf{right}}$, $\mathsf{translate} = F_{r_\ell}(1, \mathsf{right}, 1) \oplus \mathsf{label}_1^{\mathsf{right}}$. This way the evaluator can input the valid labels that corresponds to values r_m, r_{m+1} without knowing these values.[14]

The step circuit. Recall that the tree with n leafs has $d + 1$ layers: $0, \ldots, d$ $(d = \log n)$ such that the data items reside on layer d and the keys reside on the rest of the layers. In order to navigate to the L-th data item on layer d we skip the 0-layer of the tree (the root) and navigate through the rest of the $d - 1$ layers $1, \ldots, d - 1$. Next, circuit step is evaluated by choosing one out of the two data items (from level d) that are given as input.

To be concrete, circuit step_i is given 2 data items as input $D_{L+b}, D_{L+(1-b)}$ $(b \in \{0, 1\})$ along with the location L that is required for the i-th CPU-step and the current state of the program. As before, the values r and s, as well as a set of labels label_b^x (for $x \in \mathsf{left}, \mathsf{right}$, $b \in \{0, 1\}$) are hardcoded into this circuit, where r is the key that is associated with the parent of $D_{L+b}, D_{L+(1-b)}$, s is the key of the root of the tree and label_b^x are the labels for the input wires that correspond to the key pair of the next navigation circuit $\mathsf{nav}_{i+1,0}$, this is the first navigation circuit of the next CPU-step, the keys that it is given are those of locations $(1, 0)$ and $(1, 1)$ in the tree.

Circuit step_i works as follows:

– Set $b = L_{d-1}$ and $D_{\mathsf{read}} = D_{L+b}$ to be the data item that is required for the i-th CPU-step. Upon reading D_{read}, the evaluator computes $(\mathsf{state}', L', D_{\mathsf{write}}) = C_{\mathsf{CPU}}^P(\mathsf{state}, D_{\mathsf{read}})$, where state' is the new state of the program, L' is the next location to read from and D_{write} is the value to overwrite location D_{L+b} with.

– Encrypt the updated data items in locations $D_{L+b}, D_{L+(1-b)}$ (where now D_{L+b} contains D_{write}) with their parent's key r. If $b = 0$ output $\big(f_r(D_L, \mathsf{left}), f_r(D_{L+1}, \mathsf{right}), L', \mathsf{state}'\big)$, otherwise output $\big(f_r(D_{L-1}, \mathsf{left}), f_r(D_L, \mathsf{right}), L', \mathsf{state}'\big)$.

– Note that the following circuit $\mathsf{nav}_{i+1,0}$ restarts the navigation from layer 1 toward location L' (skipping layer 0). Thus, it expects to be given as input the two keys $r_{1,0}, r_{1,1}$. Those keys are encrypted by the root-key, s, which is, as mentioned earlier, is hardwired into the current step_i circuit[15]. Thus step_i also outputs a translation table $\mathsf{translate}$ that would help to input these two keys $r_{1,0}, r_{1,1}$ to the $\mathsf{nav}_{i+1,0}$ circuit.

The overall construction can be seen as a chain of T groups of circuits such that each group consists of $d - 1$ nav circuits and one step circuit. Each of the nav circuits is hardwired with a new fresh key and the

---

[14]We note that our description is informal. Specifically, the keys are encrypted bit-by-bit implying 2κ ciphertexts, as each key is of size κ. Also, the authors of [GLOS15] define a test procedure Test that can tell whether a given label is valid for some input wire or not. This way, the evaluator can identify the correct label while applying the translation procedure. For simplicity we omit these details from our description.

[15]This root key is replaced in the first navigation circuit by a new fresh key.

key that was refreshed in the prior circuit. A demonstration of a program with two CPU-steps and 8 data items is presented next.

6.1.3 Program Execution Demonstration

We start with a memory with 8 items: $D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$. Thus, the new plain data would be $D = \{r_{0,0}, r_{1,0}, r_{1,1}, r_{2,0}, r_{2,1}, r_{2,2}, r_{2,3}, D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7\}$ and the garbled data would be

$$
\tilde{D} = \Big\{ F_{r_{0,0}}(r_{1,0}, \mathsf{left}), F_{r_{0,0}}(r_{1,1}, \mathsf{right}), F_{r_{1,0}}(r_{2,0}, \mathsf{left}), F_{r_{1,0}}(r_{2,1}, \mathsf{right}), F_{r_{1,1}}(r_{2,2}, \mathsf{left}),
$$
$$
F_{r_{1,1}}(r_{2,3}, \mathsf{right}), F_{r_{2,0}}(D_0, \mathsf{left}), F_{r_{2,0}}(D_1, \mathsf{right}), F_{r_{2,1}}(D_2, \mathsf{left}), F_{r_{2,1}}(D_3, \mathsf{right}),
$$
$$
F_{r_{2,2}}(D_4, \mathsf{left}), F_{r_{2,2}}(D_5, \mathsf{right}), F_{r_{2,3}}(D_6, \mathsf{left}), F_{r_{2,3}}(D_7, \mathsf{right}) \Big\}.
$$

Let the program P consists of the instructions: $\{\mathsf{i} = x \cdot D[3]; \mathsf{output}\ \mathsf{i} \cdot D[7]; \}$. That is, we have 2 memory accesses to locations 3 and 7 and finally the program outputs $x \cdot D[3] \cdot D[7]$. Furthermore, L is of length 3 bits starting 000 and till 111 ($L = 3$ means $L = 011$ and $L = 7$ means $L = 111$). Note that the program uses an internal variable i in its state. The circuits of $\tilde{P}$ works as follows (we ignore the hardwired labels and the translation table for simplicity):

nav$_{1,0}$. **Inputs: keys** $= \{r_{1,0}, r_{1,1}\}$, $L = 3$, **state** $= x$.
Hardwired: $v_0, v_1, i = 0$.
Data $\tilde{D}$ upon navigation:

$$
\tilde{D} = \Big\{ {\color{blue}F_{v_0}(v_1, \mathsf{left}), F_{v_0}(r_{1,1}, \mathsf{right}),} F_{r_{1,0}}(r_{2,0}, \mathsf{left}), F_{r_{1,0}}(r_{2,1}, \mathsf{right}), F_{r_{1,1}}(r_{2,2}, \mathsf{left}),
$$
$$
F_{r_{1,1}}(r_{2,3}, \mathsf{right}), F_{r_{2,0}}(D_0, \mathsf{left}), F_{r_{2,0}}(D_1, \mathsf{right}), F_{r_{2,1}}(D_2, \mathsf{left}), F_{r_{2,1}}(D_3, \mathsf{right}),
$$
$$
F_{r_{2,2}}(D_4, \mathsf{left}), F_{r_{2,2}}(D_5, \mathsf{right}), F_{r_{2,3}}(D_6, \mathsf{left}), F_{r_{2,3}}(D_7, \mathsf{right}) \Big\}
$$

nav$_{1,1}$. **Inputs: keys** $= \{r_{2,0}, r_{2,1}\}$, $L = 3$, **state** $= x$.
Hardwired: $v_1, v_2, i = 1$.
Data $\tilde{D}$ upon navigation:

$$
\tilde{D} = \Big\{ F_{v_0}(v_1, \mathsf{left}), F_{v_0}(r_{1,1}, \mathsf{right}), {\color{red}F_{v_1}(r_{2,0}, \mathsf{left}), F_{v_1}(v_2, \mathsf{right}),} F_{r_{1,1}}(r_{2,2}, \mathsf{left}),
$$
$$
F_{r_{1,1}}(r_{2,3}, \mathsf{right}), F_{r_{2,0}}(D_0, \mathsf{left}), F_{r_{2,0}}(D_1, \mathsf{right}), F_{r_{2,1}}(D_2, \mathsf{left}), F_{r_{2,1}}(D_3, \mathsf{right}),
$$
$$
F_{r_{2,2}}(D_4, \mathsf{left}), F_{r_{2,2}}(D_5, \mathsf{right}), F_{r_{2,3}}(D_6, \mathsf{left}), F_{r_{2,3}}(D_7, \mathsf{right}) \Big\}
$$

step$_1$. **Inputs: Items** $= \{D_2, D_3\}$, $L = 3$, **state** $= x$.
Hardwired: v_2, v_0.
State upon running this step: state $= x \cdot D[3]$.

Data $\tilde{D}$ upon running this step:

$$\tilde{D} = \Big\{ F_{v_0}(v_1, \text{left}), F_{v_0}(r_{1,1}, \text{right}), F_{v_1}(r_{2,0}, \text{left}), F_{v_1}(v_2, \text{right}), F_{r_{1,1}}(r_{2,2}, \text{left}),$$

$$F_{r_{1,1}}(r_{2,3}, \text{right}), F_{r_{2,0}}(D_0, \text{left}), F_{r_{2,0}}(D_1, \text{right}), {\color{red}F_{v_2}(D_2, \text{left}), F_{v_2}(D_3, \text{right})},$$

$$F_{r_{2,2}}(D_4, \text{left}), F_{r_{2,2}}(D_5, \text{right}), F_{r_{2,3}}(D_6, \text{left}), F_{r_{2,3}}(D_7, \text{right}) \Big\}$$

Note: The above circuit is hardwired with v_0 and thus can decrypt the two values in level 1 of the tree, currently these values are v_1 and $r_{1,1}$. This kick starts the evaluation of the next CPU-step, which begins by the circuit $\text{nav}_{2,0}$.

$\text{nav}_{2,0}$. **Inputs:** $\text{keys} = \{v_1, r_{1,1}\}$, $L = 7$, $\text{state} = x \cdot D[3]$.
Hardwired: $u_0, u_1, i = 0$.
Data $\tilde{D}$ upon navigation:

$$\tilde{D} = \Big\{ {\color{red}F_{u_0}(v_1, \text{left}), F_{u_0}(u_1, \text{right})}, F_{v_1}(r_{2,0}, \text{left}), F_{v_1}(v_2, \text{right}), F_{r_{1,1}}(r_{2,2}, \text{left}),$$

$$F_{r_{1,1}}(r_{2,3}, \text{right}), F_{r_{2,0}}(D_0, \text{left}), F_{r_{2,0}}(D_1, \text{right}), F_{v_2}(D_2, \text{left}), F_{v_2}(D_3, \text{right}),$$

$$F_{r_{2,2}}(D_4, \text{left}), F_{r_{2,2}}(D_5, \text{right}), F_{r_{2,3}}(D_6, \text{left}), F_{r_{2,3}}(D_7, \text{right}) \Big\}$$

$\text{nav}_{2,1}$. **Inputs:** $\text{keys} = \{r_{2,2}, r_{2,3}\}$, $L = 7$, $\text{state} = x \cdot D[3]$.
Hardwired: $u_1, u_2, i = 1$.
Data $\tilde{D}$ upon navigation:

$$\tilde{D} = \Big\{ F_{u_0}(v_1, \text{left}), F_{u_0}(u_1, \text{right}), F_{v_1}(r_{2,0}, \text{left}), F_{v_1}(v_2, \text{right}), {\color{red}F_{u_1}(r_{2,2}, \text{left}),}$$

$${\color{blue}F_{u_1}(u_2, \text{right})}, F_{r_{2,0}}(D_0, \text{left}), F_{r_{2,0}}(D_1, \text{right}), F_{v_2}(D_2, \text{left}), F_{v_2}(D_3, \text{right}),$$

$$F_{r_{2,2}}(D_4, \text{left}), F_{r_{2,2}}(D_5, \text{right}), F_{r_{2,3}}(D_6, \text{left}), F_{r_{2,3}}(D_7, \text{right}) \Big\}$$

step_2. **Inputs:** $\text{Items} = \{D_6, D_7\}$, $L = 7$, $\text{state} = x \cdot D[3]$.
Hardwired: u_2, u_0.
State upon running this step: $\text{state} = x \cdot D[3] \cdot D[7]$.
Data $\tilde{D}$ upon running this step:

$$\tilde{D} = \Big\{ F_{u_0}(v_1, \text{left}), F_{u_0}(u_1, \text{right}), F_{v_1}(r_{2,0}, \text{left}), F_{v_1}(v_2, \text{right}), F_{u_1}(r_{2,2}, \text{left}),$$

$$F_{u_1}(u_2, \text{right}), F_{r_{2,0}}(D_0, \text{left}), F_{r_{2,0}}(D_1, \text{right}), F_{v_2}(D_2, \text{left}), F_{v_2}(D_3, \text{right}),$$

$$F_{r_{2,2}}(D_4, \text{left}), F_{r_{2,2}}(D_5, \text{right}), {\color{red}F_{u_2}(D_6, \text{left}), F_{u_2}(D_7, \text{right})} \Big\}$$

Where the state in the last CPU-step is outputted in the clear.

6.2 2PC in the Presence of Malicious Adversaries Relying on OWF

In order to apply the cut-and-choose technique we need to figure out what randomness affects the data to be written to the memory (because we want all copies of the circuits to output the same values). We note that the above fresh keys that are hardwired into the circuits, together with the initial data D and the program input x fully determine the values to be written to the memory during the execution of the program. This leads to the concrete construction of a malicious secure protocol using the cut-and-choose technique.

The protocol that relies on the existence of one way functions is the same as the protocol described in Section 4.1 with the differences that now the random inputs R_1, R_2 that the parties enter the computation, as well as the values that the C_{INIT} generates, are interpreted differently. We next overview the changes to the protocol from Section 4.1 in more details.

- **Interpretation of R_1 and R_2.** The circuit C_{INIT} takes R_1, R_2 as inputs, calculates $R = R_1 \oplus R_2$ and interprets R as values: $\{u_0^t, \ldots, u_{d-1}^t\}_{t \in [T]}$ which are the fresh keys that are used in the execution of the program, grouped into T parts.

- **Construction of the chains of circuits.** We now describe how the chains of circuits are being built in the new construction. To simplify notation, we describe how a single chain is being built, out of the s chains that are used in the cut-and-choose process. A chain consists of $T \cdot d + 1$ garbled circuits where the first garbled circuit is C_{INIT} followed by groups of d garbled circuits, such that the last circuit in each group t is a step circuit $step_t$ whereas the rest $d - 1$ garbled circuits are navigation circuits $nav_{t,0}, \ldots, nav_{t,d-1}$. The functionalities of these circuits were described above (see [GHL$^+$14] for further explanation). For each time step $t = T, \ldots, 1$, the garbling procedure starts by garbling $step_t$, then garbling $nav_{t,d-1}$ and so on till the garbling of $nav_{t,1}$. Such that the labels associated with the input wires of $nav_{t,j}$, that represent the current state, are hardwired into circuit $nav_{t,j-1}$ and similarly, the labels of the input wires of $step_t$, that represent the current state, are hardwired into circuit $nav_{t-1,d-1}$. This is done in the same manner as in the protocol described in Section 4.1.

- **The bootstrapping circuit.** The bootstrapping circuit (C_{INIT}) hands the values u_0^1, u_1^1 to the navigation circuit $nav_{1,0}$ (where $u_0^1 = r_{0,0}$ - the first root-key); in addition, it hands u_1^1, u_2^1, to the $nav_{1,1}$ navigation circuit and so on till the first step circuit $step_1$ which is given the keys u_{d-1}^1, u_0^1. Generally speaking, the bootstrapping circuit transfers the navigation circuit $nav_{i,j}$ the keys u_j^i, u_{j+1}^i, and the circuit $step_i$ the keys u_{d-1}^i, u_0^i.

Security intuition. The above nav and $step$ circuits are given two random keys as an input from the bootstrapping circuit C_{INIT}. These circuits relate to the keys they are given as if they were hardcoded in them, that is, the functionality remains the same. Next we argue the following two arguments: (a) Correctness: That applying the cut-and-choose technique does not require a usage of multiple instances of memory D, i.e., that the same write-data is being output from all chain copies; and (b) Security: That the evaluator does not learn anything beyond the program output and the memory access. We base these argument on the following informal arguments:

- **Correctness.** Since all the PRF keys used during the execution of the program are extracted from R_1 and R_2 that are input by the parties, it follows that if the same R_1, R_2 are being used by all copies of C_{INIT} then the same values will be output by all navigation and step circuits copies.

- **Privacy.** A simulator in this case can be constructed in the exact same manner as it was built in 5.2, with the exception that we now combine the simulators of [LP11] and [GLOS15]. The only difference

is that now, the random strings R_1, R_2 that the parties input to the chains are interpreted as the fresh random keys that are used in each of the navigation and step circuits rather then being used for the IBE procedures. We note that the random string $R = R_1 \oplus R_2$ is being hidden from the evaluator just like it was hidden in the previous construction. Also, an existence

As above, in case that the garbler has cheated, it is guaranteed by the cut-and-choose analysis that the majority of the chains will output the same value to be written to memory, with high probability.

References

[AHMR15] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In *EUROCRYPT*, pages 702–729, 2015.

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

[Bea91] Donald Beaver. Foundations of secure interactive computing. In *CRYPTO*, pages 377–391, 1991.

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

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *CCS*, pages 784–796, 2012.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In Harriet Ortiz, editor, *22nd STOC*, pages 503–513. ACM, 1990.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

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

[CR72] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, pages 73–80, 1972.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.

[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 *PETS*, pages 1–18, 2013.

[GGMP16] Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty ram computation in constant rounds. To appear in TCC, 2016.

[GHJR14] Craig Gentry, Shai Halevi, Charanjit S. Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. *IACR Cryptology ePrint Archive*, 2014:345, 2014.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, pages 405–422, 2014.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, pages 513–524, 2012.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. 2015.

[GLOS15]   Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *STOC*, pages 449–458, 2015.

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

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

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

[Gol87]   Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987.

[Gol04]   Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[HMR15]   Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In *CRYPTO*, pages 150–169, 2015.

[IKO+11]   Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In *EUROCRYPT*, pages 406–425, 2011.

[IPS08]   Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, pages 572–591, 2008.

[IPS09]   Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In *TCC*, pages 294–314, 2009.

[JS07]   Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114, 2007.

[KLO12]   Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156, 2012.

[KS14]   Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *ASIACRYPT*, pages 506–525, 2014.

[LHS+14]   Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient ram-model secure computation. In *IEEE Symposium on Security and Privacy*, pages 623–638, 2014.

[Lin13]   Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO (2)*, pages 1–17, 2013.

[LO13]   Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, pages 719–734, 2013.

[LP07]   Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, pages 52–78, 2007.

[LP11]   Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *TCC*, pages 329–346, 2011.

[Mia]   Peihan Miao. Cut-and-choose for garbled ram. Personal Communication.

[MR91]   Silvio Micali and Phillip Rogaway. Secure computation (abstract). In *CRYPTO*, pages 392–404, 1991.

[NO09]   Jesper Buus Nielsen and Claudio Orlandi. Lego for two-party secure computation. In *TCC*, pages 368–386, 2009.

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

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

[PSSW09]   Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, pages 250–267, 2009.

[RFK$^+$15] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX*, pages 415–430, 2015.

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

[SvDS$^+$13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, pages 299–310, 2013.

[WCS15]   Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In *CCS*, pages 850–861, 2015.

[WHC$^+$14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *CCS*, pages 191–202, 2014.

[WS12]   Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *CCS*, pages 293–304, 2012.

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

[Yao86]   Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

A   Building Blocks

In this section we discuss the notations and definitions of some of the standard building blocks employed in our constructions, as well as a formal description of the circuits and functionalities used in our protocols.

A.1   Garbled Circuits

The definition of garbled circuits with respect to the cut-and-choose technique is presented in Section 2.6. In this section we present the Input Consistency Functionality (Figure 7) which is realized via a secure 2PC protocol when the underlying garbling scheme is applied using a cut-and-choose based protocol. We next present the authenticity game (Figure 8) used in the definition of garbled circuits.

A.2   The Hybrid Model

The $\mathcal{F}$-hybrid model.   In order to simplify the exposition of our main protocol, we will use secure two-party protocols as subprotocols. The standard way of doing this is to work in a "*hybrid model*" where parties both interact with each other (as in the real model) and use trusted help (as in the ideal model). Specifically, when constructing a protocol π that uses a subprotocol for securely computing some functionality $\mathcal{F}$, we consider the case that the parties run π and use "ideal calls" to a trusted party for computing $\mathcal{F}$. Upon receiving the inputs from the parties, the trusted party computes $\mathcal{F}$ and sends all parties their output. Then, after receiving these outputs back from the trusted party the protocol π continues. Let $\mathcal{F}$ be a functionality and let π be a two-party protocol that uses ideal calls to a trusted party computing $\mathcal{F}$. Furthermore, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine. Then, the $\mathcal{F}$-*hybrid execution of* π on inputs (x_1, x_2), auxiliary input z to $\mathcal{A}$ and security parameter κ, denoted $\mathbf{Hyb}_{\pi^{\mathcal{F}}, \mathcal{A}(z)}(\kappa, x_1, x_2)$, is defined as the output of the honest party and the adversary $\mathcal{A}$ from the hybrid execution of π with a trusted party

<div style="border:1px solid black; padding:10px">

The Input Consistency Functionality - $\mathcal{F}_{\mathrm{IC}}$

The functionality checks that the set of garbled inputs $\{\tilde{x}_i\}_i$ that are sent to the receiver represent the same input x. Note that this functionality checks the input of the *sender* S, and thus, the variable x in this context actually refers to its input only (and not the receiver's input). Also note that $|x| = v_{\mathrm{in}}$.

Common inputs.

- The circuit C and the security parameters κ, s.

- s garbled versions of C, namely $\{\tilde{C}_i\}_{i \in [s]}$.

- s sets of garbled input $\left\{\left(\mathsf{lbl}^{1,i}_{\mathrm{in},x[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{in}},i}_{\mathrm{in},x[v_{\mathrm{in}}]}\right)\right\}_{i \in [s]}$.

- s sets of commitments for the sender's input labels, denoted by $\{\mathsf{com}^i_{1,b}, \ldots, \mathsf{com}^i_{v_{\mathrm{in}},b}\}_{b \in \{0,1\}, i \in [s]}$.

Sender's private inputs.   (The receiver has no private input)

- The output labels used in Garb, denoted by $\{\mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b}$.

- The randomness r used in Garb.

- Decommitments $\{\mathsf{dec}^i_{1,b}, \ldots, \mathsf{dec}^i_{v_{\mathrm{in}},b}\}_{b \in \{0,1\}, i \in [s]}$ for the above commitments to the input labels.

Output.   The functionality works as follows:

- Compute
$$\left(\{\hat{C}_i\}_i, \{u, b, \hat{\mathsf{lbl}}^{u,i}_{\mathrm{in},b}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, C, \{v, b, \mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b}; r\right)$$

- For every $u \in [v_{\mathrm{in}}]$:

  - For every $i \in [s]$ set b_i as
  $$b_i = \left\{ \begin{array}{ll} 0, & \mathsf{com}(\mathsf{lbl}^{u,i}_{\mathrm{in},x[u]}, \mathsf{dec}^i_{u,0}) = \mathsf{com}^i_{u,0} \\ 1, & \mathsf{com}(\mathsf{lbl}^{u,i}_{\mathrm{in},x[u]}, \mathsf{dec}^i_{u,1}) = \mathsf{com}^i_{1,0} \\ \bot & \text{otherwise} \end{array} \right\}$$

  - If $b_i = \bot$ for some i then output 0. Also If $b_1 \neq b_i$ for some i output 0. (This checks that all labels are interpreted as the same input bit in all garbled circuits).

- Given that the above algorithm has not output 0, then output 1.

</div>

Figure 7: The input consistency functionality $\mathcal{F}_{\mathrm{IC}}$.

computing $\mathcal{F}$. By the composition theorem [Can00] any protocol that securely implements $\mathcal{F}$ can replace the ideal calls to $\mathcal{F}$.

A.3 Batch Single-Choice Cut-and-Choose OT

The Batch Single-Choice Cut-and-Choose Oblivious Transfers is presented in Figure 9

The authenticity game $\mathrm{Auth}_{\mathcal{A}}(1^\kappa, s, \mathrm{C})$

Parameters. For an arbitrary circuit C, a security parameters κ and s the game is as follows.

1. The adversary hands an input x and a subset $Z \in [s]$ to the game.

2. The game chooses s sets of output labels $\{v, b, \mathsf{lbl}_{\mathrm{out},b}^{v,i}\}_{v,i,b}$ for every $v \in [v_{\mathrm{out}}], i \in [s]$ and $b \in \{0,1\}$ and computes:

$$\left(\{\tilde{\mathrm{C}}_i\}_i, \{u, b, \mathsf{lbl}_{\mathrm{in},b}^{u,i}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, \mathrm{C}, \{v, b, \mathsf{lbl}_{\mathrm{out},b}^{v,i}\}_{v,i,b}\right)$$

3. The game sends the adversary s sets of garbled circuits $\{\tilde{\mathrm{C}}_i\}_{i \in [s]}$ and s sets of garbled inputs $\tilde{x}_i$: For garbled circuits indexed with $z \in Z$ it is given $\tilde{x}_z = (\mathsf{lbl}_{\mathrm{in},b}^{1,z}, \ldots, \mathsf{lbl}_{\mathrm{in},b}^{v_{\mathrm{in}},z})$ for every $b \in \{0,1\}$; while for garbled circuits indexed with $z \notin Z$ the set is $\tilde{x}_z = (\mathsf{lbl}_{\mathrm{in},x[1]}^{1,z}, \ldots, \mathsf{lbl}_{\mathrm{in},x[v_{\mathrm{in}}]}^{v_{\mathrm{in}},z})$ for some input x.

4. The adversary returns a single index z and one set of output labels $\hat{y}_z = (\hat{\mathsf{lbl}}_{\mathrm{out},b}^{1,z}, \ldots, \hat{\mathsf{lbl}}_{\mathrm{in},b}^{v_{\mathrm{out}},z})$.

5. The game concludes as follows:

   (a) If $z \in Z$ return 0. Otherwise continue.
   (b) Compute $(\mathsf{lbl}_{\mathrm{out},y[1]}^{1,z}, \ldots, \mathsf{lbl}_{\mathrm{in},y[v_{\mathrm{out}}]}^{v_{\mathrm{out}},z}) = \mathsf{Eval}(\tilde{\mathrm{C}}_z, \tilde{x}_z)$.
   (c) If for some $j \in [v_{\mathrm{out}}]$ it holds that $\hat{\mathsf{lbl}}_{\mathrm{out},b}^{j,z} = \mathsf{lbl}_{\mathrm{out},1-y[1]}^{1,z}$ then output 1. Otherwise, output 0.

Figure 8: The authenticity game $\mathrm{Auth}_{\mathcal{A}}(1^\kappa, s, \mathrm{C})$.

Batch Single Choice Cut-And-Choose OT $\mathcal{F}_{\mathrm{SCCOT}}$

Inputs.

- The sender S inputs vectors of pairs $\mathbf{x}_i$ of length s, for $i = 1, \ldots, \ell$ (where ℓ is the input length of the parties, i.e. $\ell = |x_i| + |R_i|$ for $i \in \{1,2\}$). Every vector is a row of s pairs. There are ℓ such rows. This can be viewed as an $s \times \ell$ matrix of pairs).

- The receiver R inputs $x_2[1], \ldots, x_2[\ell] \in \{0,1\}$ and a set of indices $Z \subset [s]$ of size exactly $s/2$. (For every row the receiver chooses a bit σ_i. It also chooses $s/2$ of the s columns.)

Output. If Z is not of size $s/2$ then S and R receive for output $\perp$. Otherwise,

- For every $j = 1, \ldots, \ell$ and for every $z \in Z$, the receiver R obtains the zth pair in vector $\mathbf{x}_j$. (i.e. the receiver obtains the *two* items of every pair, in all rows.)

- For every $j = 1, \ldots, \ell$ and for every $z \notin Z$ the receiver R obtains the $x_2[j]$ value in every pair of the vector $\mathbf{x}_i$. (i.e. the receiver obtains its choice $x_2[j]$ of the two items in the pair, where $x_2[j]$ is the same for all entries in a row.)

Figure 9: Batch single choice cut-and-choose OT $\mathcal{F}_{\mathrm{SCCOT}}$.

A.4 Commitment Schemes

Commitment schemes are used to enable a party, known as the *sender*, to commit itself to a value while keeping it secret from the *receiver* (this property is called *hiding*). Furthermore, in a later stage when the commitment is opened, it is guaranteed that the "opening" can yield only a single value determined in the committing phase (this property is called *binding*). In this work, we consider commitment schemes that are *statistically-binding*, namely while the hiding property only holds against computationally bounded (non-uniform) adversaries, the binding property is required to hold against unbounded adversaries.

Definition A.1 (Commitment schemes.). *A pair of* PPT *machines* $\mathsf{Com} = (\mathrm{R}, \mathrm{S})$ *is said to be a commitment scheme if the following two properties hold.*

Computational hiding: *For every (expected)* PPT *machine* R^**, it holds that, the following ensembles are computationally indistinguishable.*

- $\{\mathbf{View}_{\mathsf{Com}}^{\mathrm{R}^*}(m_1, z)\}_{n \in \mathbb{N}, m_1, m_2 \in \{0,1\}^n, z \in \{0,1\}^*}$
- $\{\mathbf{View}_{\mathsf{Com}}^{\mathrm{R}^*}(m_2, z)\}_{n \in \mathbb{N}, m_1, m_2 \in \{0,1\}^n, z \in \{0,1\}^*}$

where $\mathbf{View}_{\mathsf{Com}}^{\mathrm{R}^*}(m, z)$ *denotes the random variable describing the output of* R^* *upon interacting with the sender* S *which commits to* m*.*

Statistical binding: *Informally, the statistical-binding property asserts that, with overwhelming probability over the coin-tosses of the receiver* R*, the transcript of the interaction fully determines the value committed to by the sender.*

Formally, a receiver's view of an interaction with the sender, denoted $(r, \bar{m})$*, consists of the random coins used by the receiver (namely,* r*) and the sequence of messages received from the receiver (namely,* $\bar{m}$*). Let* $m_1, m_2 \in \mathcal{M}_n$*. We say that the receiver's view (of such interaction),* $(r, \bar{m})$*, is* a possible m-commitment *if there exists a string* s *such that* $\bar{m}$ *describes the messages received by* R *when* R *uses local coins* r *and interacts with* S *which uses local coins* s *and has input* $(1^n, m)$*.*

We say that the receiver's view $(r, \bar{m})$ *is* ambiguous *if is it both a possible* m_1-commitment *and a possible* m_2-commitment*. The binding property asserts that, for all but a negligible fraction of the coins toss of the receiver, there exists no sequence of messages (from the sender) which together with these coin toss forms an ambiguous receiver view. Namely, that for all but a negligible function of the* $r \in \{0,1\}^{\mathsf{poly}(n)}$ *there is no* $\bar{m}$ *such that* $(r, \bar{m})$ *is ambiguous.*

B Realizing the Garbled Circuit Definition 2.6

In this section we argue that our definition of garbled circuit with respect to cut-and-choose based protocols (see Definition 2.6) can be realized by the [LP11] garbling scheme. We first describe the algorithms Garb, Eval and then argue that they posses the *correctness*, *privacy*, *authenticity* and *input consistency* properties.

Garbling. Recall that in the notion of cut-and-choose based protocols the garbling scheme is given s sets of output labels, from which it has to produce s garbled circuits along with their corresponding garbled inputs. To simplify notation, we first describe in Figure 10 the garbling procedure for a *single* circuit; then, in Figure 11 we describe the full garbling procedure Garb that uses GarbYao as a sub-procedure.

Evaluation. As modulated in the garbling procedure, we first show how a single garbled circuit can be evaluated in Figure 12 and then in Figure 13 we show how, using EvalYao as a sub-procedure, we evaluate a set of s garbled circuits.

Correctness and privacy. The correctness and privacy properties had been proven in [LP07] for a single circuit. It is trivial to show that these properties hold in the cut-and-choose notion we defined.

Procedure GarbYao

Input.

- A circuit description C_{CPU} with v_{in} (resp. v_{out}) input (resp. output) wires, and a total of W wires.

- A set of v_{out} output labels $\mathsf{lbl}^0_{\text{out},1}, \mathsf{lbl}^1_{\text{out},1}, \ldots, \mathsf{lbl}^0_{\text{out},v_{\text{out}}}, \mathsf{lbl}^1_{\text{out},v_{\text{out}}}$.

- A set of input labels $\mathsf{lbl}^0_{\text{in},1}, \mathsf{lbl}^1_{\text{in},1}, \ldots, \mathsf{lbl}^0_{\text{in},v_{\text{in}}}, \mathsf{lbl}^1_{\text{in},v_{\text{in}}}$.

Output.

- Choose a pair of random labels $\mathsf{lbl}^0, \mathsf{lbl}^1$ for every wire in W that is neither an input nor output wire.

- Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme that has indistinguishable encryptions for multiple messages, and has an elusive efficiently verifiable range. (cf. [LP07]). For each gate $G \in C_{\text{CPU}}$ with input wires a, b and output wire c such that G computes the binary function $g : \{0,1\}^2 \to \{0,1\}$, compute $\tilde{G} = (\mathsf{ct}_{00}, \mathsf{ct}_{01}, \mathsf{ct}_{10}, \mathsf{ct}_{11})$ where

$$\mathsf{ct}_{00} = \mathsf{Enc}_{\mathsf{lbl}^0_a}(\mathsf{Enc}_{\mathsf{lbl}^0_b}(\mathsf{lbl}^{g(00)}_c)) \quad , \quad \mathsf{ct}_{01} = \mathsf{Enc}_{\mathsf{lbl}^0_a}(\mathsf{Enc}_{\mathsf{lbl}^1_b}(\mathsf{lbl}^{g(01)}_c))$$

$$\mathsf{ct}_{10} = \mathsf{Enc}_{\mathsf{lbl}^1_a}(\mathsf{Enc}_{\mathsf{lbl}^0_b}(\mathsf{lbl}^{g(10)}_c)) \quad , \quad \mathsf{ct}_{11} = \mathsf{Enc}_{\mathsf{lbl}^1_a}(\mathsf{Enc}_{\mathsf{lbl}^1_b}(\mathsf{lbl}^{g(11)}_c))$$

- Return $\widetilde{C_{\text{CPU}}} = \bigcup_{G \in C_{\text{CPU}}} \tilde{G}$.

Figure 10: Procedure GarbYao for a single circuit garbling.

Authenticity. The authenticity property is missing from [GHL$^+$14] while it is indeed required even in the semi-honest model. We now show that the above scheme has authenticity. Informally, breaking authenticity means that the evaluator guesses a secret label that is not in the encoded output. Switching to a simulated garbling the way defined in [LP07] produces an indistinguishable view, in that case the probability of guessing an additional label is negligible since the inactive labels are not used at all, then it should be that case for garbled circuits as well.

More formally, given a circuit C_{CPU} and an adversary $\mathcal{A}$, for which $\Pr[\text{Auth}_{\mathcal{A}}(1^\kappa, s, C) = 1] = p$, we construct a distinguisher $\mathcal{D}$ for the simulator SimGC that succeed in distinguishing with the same probability. $\mathcal{D}$ is given a view which contains the garbled circuit $\widetilde{C_{\text{CPU}}}$ and a garbled input $\tilde{x}$ for a given input x such that $\widetilde{C_{\text{CPU}}}(\tilde{x}) = \tilde{y}$. $\mathcal{D}$ hands $\widetilde{C_{\text{CPU}}}, x, \tilde{x}$ to $\mathcal{A}$, if $\mathcal{A}$ outputs a valid $\hat{a}$ then output 1, otherwise output 0. Note that the probability that $\mathcal{A}$ outputs a valid $\hat{y}$ when given a simulated view is negligible ϵ (since the inactive labels are merely random string), thus if $\mathcal{A}$ outputs a valid $\hat{y}$ it means that it got a real view with probability $p - \epsilon$. If p is non-negligible then $\mathcal{D}$ succeeds in distinguishing with non-negligible probability.

Input consistency. We now show the protocol that realizes the input consistency functionality $\mathcal{F}_{\text{IC}}$ from Figure 7 with respect to the garbling scheme $(\mathsf{Garb}, \mathsf{Eval})$ from above. The common inputs are

- Security parameters s, κ.

- The circuit C_{CPU} and s garbled versions $\{\widetilde{C_{\text{CPU}}}_i\}_{i \in [s]}$. Note that s here is a subset of the s which the sender used in the garbling phase.

- Labels $(\mathsf{lbl}^i_1, \ldots, \mathsf{lbl}^i_\ell) = (g^{a^{x[1]}_1 \cdot r_i}, \ldots, g^{a^{x[\ell]}_\ell \cdot r_i})$ for all i.

- Commitments to all the sender's input labels: $a^0_1, a^1_1, \ldots, a^0_\ell, a^1_\ell \in \mathbb{Z}_q$ and $r_1, \ldots, r_\ell \in \mathbb{Z}_q$.

Protocol. The sender proves that for every $j \in [\ell]$ the set $\{g^{a^{x[j]}_j \cdot r_i}\}_{i \in [s]}$ is consistent: For every $j \in [\ell]$ the sender uses the protocol in Figure 14 to prove that there exist a value $\sigma_j \in \{0,1\}$ such that for every $i \in [s]$, $\mathsf{lbl}^i_j = g^{a^{\sigma_j}_j \cdot r_i}$.

Procedure Garb

Parameters. A "fixed" parameters for the garbling is a description $(\mathbb{G}, q, g)$ where $\mathbb{G}$ is a cyclic group with generator g and prime order q. **Inputs.**

- Security parameters $s, 1^\kappa$

- A circuit description C_{CPU} with v_{in} (resp. v_{out}) input (resp. output) wires, and a total of W wires.

- s set of v_{out} output labels $\mathsf{lbl}^{0,i}_{\text{out},1}, \mathsf{lbl}^{1,i}_{\text{out},1}, \ldots, \mathsf{lbl}^{0,i}_{\text{out},v_{\text{out}}}, \mathsf{lbl}^{1,i}_{\text{out},v_{\text{out}}}$ for all $i \in [s]$.

Output.

- Let $[\ell] \subset [v_{\text{in}}]$ be the indices of input wires that are associated with the sender's input.

- Choose $a^0_1, a^1_1, \ldots, a^0_\ell, a^1_\ell \in \mathbb{Z}_q$ and $r_1, \ldots, r_\ell \in \mathbb{Z}_q$.

- For every $j \in [\ell]$ and $i \in [s]$ set:

$$\mathsf{lbl}^{0,i}_{\text{in},j} = H(g^{a^0_j \cdot r_i}) \text{ and } \mathsf{lbl}^{1,i}_{\text{in},j} = H(g^{a^1_j \cdot r_i})$$

- Choose a pair of random labels for each of the other ℓ input wires (for the receiver's input wires), denoted by $\mathsf{lbl}^{b,i}_{\text{in},j}$ for every $j \in [\ell], b \in \{0,1\}, i \in [s]$.

- For $i = 1, \ldots, s$ compute:

$$\widetilde{C_{\text{CPU}i}} = \mathsf{GarbYao}\big(C, \big\{ \mathsf{lbl}^{b,i}_{\text{out},1}, \ldots, \mathsf{lbl}^{b,i}_{\text{out},v_{\text{out}}} \big\}_{b \in \{0,1\}}, \big\{ \mathsf{lbl}^{b,i}_{\text{in},1}, \ldots, \mathsf{lbl}^{b,i}_{\text{in},v_{\text{in}}} \big\}_{b \in \{0,1\}}\big)$$

- Return $\big\{ \widetilde{C_{\text{CPU}i}} \big\}_{i \in [s]}, \big\{ \mathsf{lbl}^{b,i}_{\text{in},1}, \ldots, \mathsf{lbl}^{b,i}_{\text{in},v_{\text{in}}} \big\}_{b \in \{0,1\}, i \in [s]}$.

Figure 11: Procedure Garb for s circuits garbling.

Namely, it proves that all garbled values of a wire are of the same bit. If any of the proofs fail, then P2 aborts and outputs $\bot$.

For completeness, we provide the protocol, used in [LP11], verbatim.

ZK proof for extended Diffie-Hellman tuples. A zero-knowledge proof of an extended Diffie-Hellman tuple is given in Figure 14. The input is a tuple $(g, h_0, h_1, u_1, v_1, \ldots, u_\eta, v_\eta)$ such that either all $\{(g, h_0, u_i, v_i)\}^\eta_{i=1}$ are Diffie-Hellman tuples, or all $\{(g, h_1, u_i, v_i)\}^\eta_{i=1}$ are Diffie-Hellman tuples. It is shown in [LP11] that the protocol in Figure 14 is a ZK-PoK.

Procedure EvalYao

Inputs.

- Garbled circuit $\widetilde{C_{\text{CPU}}}$.
- A set of v_{in} labels $\text{lbl}^1_{\text{in},x[1]}, \ldots, \text{lbl}^{v_{\text{in}}}_{\text{in},x[v_{\text{in}}]}$ for some input x.

Output.

- For every garbled gate in the set of garbled gates in $\widetilde{C_{\text{CPU}}}$ (in a topological order). Let a, b be its input wires, $\text{lbl}^\alpha_a, \text{lbl}^\beta_b$ the labels for these wires and α, β the bits $\{0, 1\}$ they represent, finally let c be G's output wire.

    - For $t = 00, 01, 10, 11$ compute $\text{pt}_t = \text{Dec}_{\text{lbl}^\beta_b}(\text{Dec}_{\text{lbl}^\alpha_a}(\text{ct}_t))$.

    - Set $\text{lbl}^\gamma_c = \text{pt}_t$ for the only t for which $\text{pt}_t \neq \bot$. (There is always a single pt_t except with negligible probability).

- Output $(\text{lbl}^1_{\text{out},y[1]}, \ldots, \text{lbl}^{v_{\text{out}}}_{\text{out},y[v_{\text{out}}]})$ for some output y.

Figure 12: Procedure EvalYao for a single garbled circuit.

Procedure Eval

Inputs.

- s garbled circuits $\{\widetilde{C_{\text{CPU}i}}\}_{i \in [s]}$.
- s sets of v_{in} labels $\left\{ (\text{lbl}^{1,i}_{\text{in},x[1]}, \ldots, \text{lbl}^{v_{\text{in}},i}_{\text{in},x[v_{\text{in}}]}) \right\}_{i \in [s]}$ for some input x.

Output.

- For every $i = 1, \ldots, s$ compute

$$(\text{lbl}^{1,i}_{\text{out},x[1]}, \ldots, \text{lbl}^{v_{\text{out}},i}_{\text{out},y[v_{\text{out}}]}) \leftarrow \text{EvalYao}\left(\widetilde{C_{\text{CPU}}}, (\text{lbl}^{1,i}_{\text{in},x[1]}, \ldots, \text{lbl}^{v_{\text{in}},i}_{\text{in},x[v_{\text{in}}]})\right)$$

- Output $\left\{ (\text{lbl}^{1,i}_{\text{out},y[1]}, \ldots, \text{lbl}^{v_{\text{out}},i}_{\text{out},y[v_{\text{out}}]}) \right\}_{i \in [s]}$ for some output y.
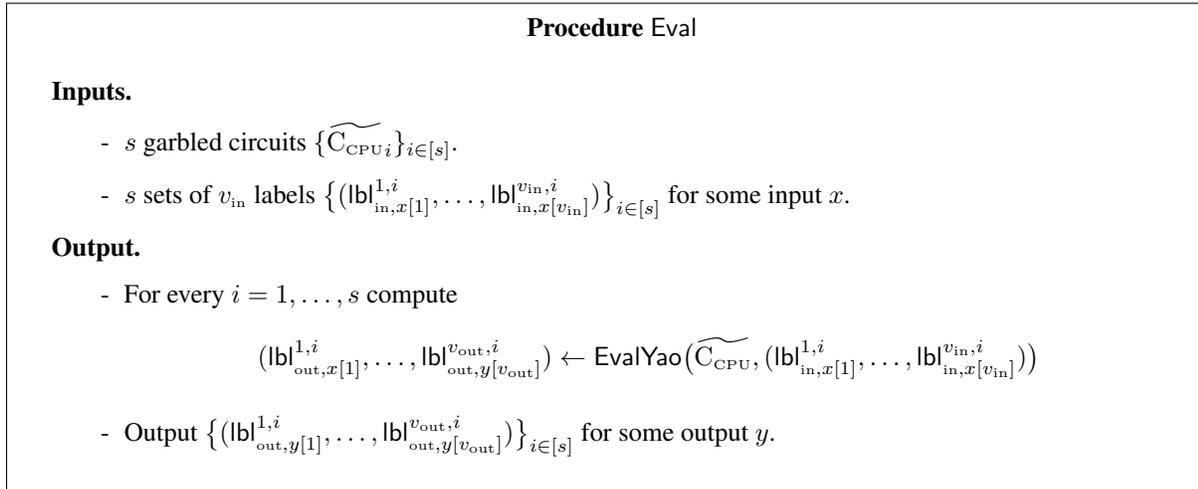
Figure 13: Procedure Eval to evaluate s garbled circuits.

ZK Proof of Knowledge of Extended Diffie-Hellman Tuple

Common input. $(g, h_0, h_1, u_1, v_1, \ldots, u_\eta, v_\eta)$ where g is a generator of a group of order q.
Prover witness. a such that either $h_0 = (g_0)^a$ and $v_i = (u_i)^a$ for all i, or $h_1 = (g_1)^a$ and $v_i = (u_i)^a$ for all i.
The protocol.

- The verifier V chooses $\gamma_1, \ldots, \gamma_\eta \in_R \{0, 1\}^L$ where $2^L < q$, and sends the values to the prover.

- The prover and verifier locally compute:

$$u = \prod_{i=1}^{\eta} (u_i)^{\gamma_i} \quad \text{and} \quad v = \prod_{i=1}^{\eta} (v_i)^{\gamma_i}$$

- The prover proves in zero-knowledge that either (g_0, h_0, u, v) or (g_1, h_1, u, v) is a Diffie-Hellman tuple, and V accepts if and only it accepts in the 1-out-of-2 ZK proof. (see [LP11] for more details).

Figure 14: ZK Proof of knowledge of extended Diffie-Hellman tuples.