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. [18] 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. [15] that is based on one-way functions.

[*]Supported by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS) and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Ministers Office. First author's research partially supported by a grant from the Israel Ministry of Science and Technology (grant No. 3-10883). An extended abstract of this paper will appear in the proceeding of TCC 2016-B.

Contents

1 Introduction

1.1 Background – Secure Computation and the RAM Model.

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 [51, 21, 2, 39, 7], 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, a lot of effort has been made into improving the efficiency of computation with the aim of minimizing the workload of the parties [29, 33, 27] [28, 42, 40, 35, 26, 32]. 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 $O(T^3 \log T)$ which are non-scalable even for cases where the memory size is relatively small [10, 43].

To address these limitations, researchers have recently started to design secure protocols directly in the RAM model [11, 24, 1]. The main underlying idea is to rely on Oblivious RAM (ORAM) [19, 41, 22], a fundamental tool that supports dynamic memory access with poly-logarithmic 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 and 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 (where a block is the atomic accessible data item in memory). 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 read/write accesses 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 [24, 16, 37, 17, 18, 30, 36, 48, 47, 1, 15, 25, 14, 12]. 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 [1] 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 with round complexity that depends on the ORAM running time. In [24] Gordon et al. designed the first provably secure semi-honest protocol based on this approach, which 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. [1], 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 prior 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. Since we are interested in concrete efficiency, we rule out practically inefficient solutions that rely on general zero-knowledge proofs or alternatively require public-key operations for every gate in the circuit. To be precise, we restrict our attention to protocols that can be described in the OT-hybrid and only rely one-way functions with polylogarithmic amortized communication overhead in the memory size of the RAM program.

1.2 Background – Garbled RAM and Circularity

The feasibility of constant round semi-honest secure two-party computation has established by Lu and Ostrovsky in [37] by introducing a new cryptographic primitive, analogue to garbled circuits [51, 34], known as Garbled RAM (or GRAM).

The Lu-Ostrovsky construction. In order to understand the difficulty in destining GRAMs we consider a simplified version in which the memory is read-only. Then the garbled data in [37], denoted by $\widetilde{D}$, consists of n secret keys for some symmetric-key encryption scheme. Namely, for each bit $i \in [n]$, $\widetilde{D}$ contains a secret key sk_i such that $\mathsf{sk}_i = F_k(i, D[i])$ and F is a pseudorandom function (PRF).

Furthermore, the garbled program $\widetilde{P}$ consists of T garbled copies of a CPU-step circuit that takes as input the current CPU state and the last read bit $(\mathsf{state}, b^{\mathsf{read}})$ and outputs $(\mathsf{state}', i^{\mathsf{read}})$ which contains the updated state and the next read location. The garbled circuit of the jth CPU-step copy is defined so that the output labels for the wires corresponding to state' match the input labels corresponding to the input state for the garbled circuit of the $(j+1)$th CPU-step copy. This allows the garbled state to be securely transferred from one garbled CPU-step circuit to another, whereas the read location i^{read} is output in the clear (assuming the running program is the product of an ORAM compiler).

It remains to incorporate the data from the memory into the computation. Let $\mathsf{lbl}_0^{(\mathsf{read},j+1)}, \mathsf{lbl}_1^{(\mathsf{read},j+1)}$ be the two input labels of the wire corresponding to the bit b^{read} within the $(j+1)$th CPU-step copy. Note that these labels are created at "compile time" whenever the garbled program is created and therefore cannot depend on i^{read} which is only known at "run time".

In order to ensure that the evaluator can only learn one of these labels, Lu and Ostrovsky devised an "augmenting" circuit where the jth CPU-step circuit outputs a *translation mapping* $\mathsf{translate}$, which allows the evaluator to translate a secret key into an input label. This translation mapping consists of two ciphertexts $\mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1)$ where ct_b is an encryption of the label $\mathsf{lbl}_b^{(\mathsf{read},j+1)}$ under the secret key $F_k(i, b)$. This requires that the augmented CPU-step circuits will be hardcoded with the PRF key k.

The circularity problem. Assume that the evaluator only gets one label per wire for the first garbled circuit (namely, $j = 1$) and therefore does not learn anything beyond $i^{\mathsf{read}}, \mathsf{translate} = (\mathsf{ct}_0, \mathsf{ct}_1))$ and the garbled value state_2 which is used as an input to the second circuit. Now, assume that $D[i^{\mathsf{read}}] = 0$ and so the evaluator can use $F_k(i^{\mathsf{read}}, 0)$ to recover the label $\mathsf{lbl}_0^{(\mathsf{read},2)}$ for the next CPU-step circuit where $j = 2$. Next, we need to argue that the evaluator does not learn anything about label $\mathsf{lbl}_1^{(\mathsf{read},2)}$. Intuitively, the above should hold since the evaluator does not know the secret key generated by $F_k(i^{\mathsf{read}}, 1)$ that is needed to decrypt ct_1. Unfortunately, attempting to make this intuition formal uncovers a complex circularity:

1. In order to argue that the evaluator does not learn anything about the "other label $\mathsf{lbl}_1^{(\mathsf{read},2)}$, we need to rely on the privacy of ciphertext ct_1.

2. In order to rely on the privacy of ciphertext ct_1 we need to argue that the attacker does not learn the secret key $F_k(i^{\mathsf{read}}, 1)$, which implies that the attacker should not use the PRF key k. However, this key is hardcoded within the second garbled circuit as well as within all future circuits. Therefore, to argue that the attacker does not use k we need to rely on the privacy of the second garbled circuit.

3. In order to rely on the privacy of the second garbled circuit we need to argue that the evaluator only learns one label per wire, and in particular, we need to argue that the evaluator does not learn the "other" label $\mathsf{lbl}_1^{(\mathsf{read},2)}$, which is what we needed to prove in the first place.

Applying our techniques to the Lu-Ostrovsky construction. Unfortunately, our techniques of factoring out the "hardcoded secrets" do not solve the above circularity problem. To illustrate that, consider the Lu-Ostrovsky construction with a small modification. Namely, the PRF k is given as an input to the first garbled circuit ($j = 1$) such that k is transmitted from one CPU-step circuit to another, just like the program state. Then it is simple to verify that the above circularity still holds. Specifically, to argue that the evaluator does not learn $\mathsf{lbl}_1^{(\mathsf{read},2)}$ we need to rely on the privacy of ct_1, which implies that we need to rely on the pseudorandomness of $F_k(i^{\mathsf{read}}, 1)$. Nevertheless, k is also transmitted to the second garbled circuit ($j = 2$), which means that security must rely on the privacy of the second garbled circuit, which again requires to rely on the fact that the evaluator does not learn $\mathsf{lbl}_1^{(\mathsf{read},2)}$ and so on.

1.3 Our Results

In this work 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. [18] that is analogous to garbled circuits [51, 4] with respect to RAM programs. Namely, a user can garble an arbitrary RAM program directly without transforming 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, that is based on identity-based encryption (IBE) schemes, Gentry et al. proposed a way to remove the circularity assumption that is additionally required in the construction of Lu and Ostrovsky [37]. We first show how to transform their IBE based protocol into a maliciously secure 2PC protocol using the cut-and-choose technique. Following that, we apply our transformation to the garbled RAM construction of Garg et al. [15] to obtain a construction under the weaker assumption of one-way functions. As a side remark, we believe that our techniques are applicable to the GRAM constructions of [37] and [14] as well. Nevertheless, we chose not to explore these directions in this work due to the non standard circularity assumption in [37] and the complicated machinery in [14].

Let $\mathrm{C}_{\mathrm{CPU}}^{P}$ be the circuit that computes a single CPU-step (which involves reading/writing to the memory), T be the upper bound on the running time of a program P on input of length $|x|$ and κ, s be the computational and statistical security parameters. Then our first main theorem states the following,

Theorem 1.1 (Informal)**.** *Assuming oblivious transfer and IBE, there exists a constant-round two-party protocol that securely realizes any RAM program in the presence of malicious adversaries, where the size of the garbled database is $n \cdot \mathsf{poly}(\kappa, \log n)$, the size of the garbled input is $|x| \cdot O(\kappa)$ and the size of the garbled program is $T \cdot \mathsf{poly}(\kappa, \log n) \cdot s$, and its evaluation time is $T \cdot \mathsf{poly}(\kappa) \cdot \mathsf{polylog}\ (n) \cdot s$.*

We next demonstrate how to apply our approach to the GRAM from [15] that is only based on one-way functions. This implies the following theorem,

Theorem 1.2 (Informal)**.** *Assuming oblivious transfer, there exists a constant-round two-party protocol that securely realizes any RAM program in the presence of malicious adversaries where the asymptotic complexities are as implied by Theorem 1.1.*

Challenges faced in the malicious setting for 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 lifting the garbled RAM's security from semi-honest to malicious is using the cut-and-choose technique [33]. This means that the basic semi-honest protocol is instantiated s times (for some statistical parameter s) and then the parties prove that they followed the prescribed protocol in a subset of the instances (this subset is chosen uniformly). Finally the parties use the remaining instances to obtain the output (typically by taking the majority of results). It has been proven that the output of this process leads to the correct output with overwhelming probability. Applying the cut-and-choose technique to the RAM model naively leads to handling multiple instances of memory. 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 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 rollback the stored memory to an earlier version. This task is very challenging in a scenario 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 (Section 4). Towards achieving malicious security, we demonstrate how to adapt the garbled RAM construction from [18] into the two-party setting while achieving malicious security. Our protocol consists 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 in the sense of definition 2.2 and can be optimized using all known optimizations (e.g. row-reduction, free-XOR, etc.). 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 [4]. 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.

Removing the IBE assumption (Section 5). Our techniques are also applicable with respect to the GRAM from [15]. Loosely speaking, in order to avoid circularity, Garg et al. considered a different approach by constructing a tree for which the memory is associated with its leafs. Moreover, each internal node is associated with a PRF key that is encrypted under a PRF key associated with this node's parent. Then, during the evaluation, each navigation circuit outputs a translation table that allows the evaluator to learn the input label for the next node based on the path to the read position in the memory. In addition, the circuit refreshes the PRF key associated with this node and computes a new set of PRF values based on this new key. This technique incurs an overhead of $\log n$ on the running time of the program since for each memory access the evaluator has to traverse a tree of depth $\log n$ and only then perform the actual access. Consequently, while our first construction based on IBE (see Sections 3 and 4) requires s chains of CPU-step circuits of size T, removing the IBE assumption implies that each CPU-step circuit is now expanded to $\log n$ navigation circuits. Moreover, the initialization circuit now generates $\log n$ fresh keys for the $\log n$ navigation circuits of each chain and passes these the keys over the input wires, which means that the initialization circuit is now of size $O(T \log n)$. On the other hand, the complexity of the initialization circuit is much simpler now as it does not need to employ the complex TIBE algorithms but rather simple XOR operations.

Complexity. The overhead of our first protocol (cf. Section 4) is dominated by the complexity induced by the garbled RAM construction of [18] times s, where s is the cut-and-choose statistical parameter. The [18] construction guarantees that the size/evaluation time of the garbled program is $|\mathrm{C}^{P}_{\mathrm{CPU}}| \times T \times \mathsf{poly}(\kappa) \times \mathsf{polylog}\ (n) \times s$. Therefore the multiplicative overhead of our protocol is $\mathsf{poly}(\kappa) \times \mathsf{polylog}\ (n) \times s$. Our second protocol (cf. Section 5), that is based on the GRAM from [15], is $\log n$ times slower than the first protocol due to the way the memory is being accessed (i.e. by traversing a tree). This has an impact on the initialization circuit of each CPU-step circuits chain. As mentioned above, the initialization circuit in the first protocol needs to realize the IBE algorithms which contribute $T \cdot \mathsf{poly}(\log k)$ to the circuit's size, whereas the initialization circuit in the second protocol needs to generates $T \cdot \log n$ random PRF keys, each of size k. The overall complexities are given in Table 1.3.

Reusable/persistent data. Reusable/persistent data means that the garbled memory data can be reused across multiple program executions. That is, all memory updates 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 amortized running time is only proportional to the running time of the program in a unsecured environment. The garbled RAM in [18] allows to garble any sequence of programs and inputs. Nevertheless, the set of programs and inputs must be determined in advance and cannot be chosen adaptively based on prior iterations. This is due to an issue, which arises in the proof, related to another open problem known as "adaptive Yao" where the evaluator of the garbled circuit may choose its input based on the garbled circuit. In this work we prove that our scheme preserves the weaker property as in [18] in the presence of malicious attacks as well.

Concurrent work. In a concurrent and independent work by Garg, Gupta, Miao and Pandey [13], the authors demonstrate constant-round *multi-party* constructions for both the semi-honest and the malicious settings. Their maliciously secure construction is based on the black-box Garbled RAM [14], the BMR constant-round MPC protocol [3] and the IPS compiler [27]. Their semi-honest secure protocol achieves persistent data, whereas their maliciously secure protocol achieves the weaker notion of selectively choosing the programs and inputs in advance, as we do. The core technique of pulling the secrets out of the programs is common to both our and their work. In contrast to [13], in our scheme only one party locally stores the memory throughout the evaluation. These differences are summarized in Table 1.3.

The parameters p, κ and s refer to the number of parties and the respective computational and statistical security parameters. We denote the ORAM's runtime and memory overheads by $\log^a n$ and $\log^b n$ respectively where a, b are constants determined by the choice of ORAM construction. The memory size in [13] has the term $O(n \log^b n + T)$ since this is based on the GRAM of [14] in which a memory entry is represented by a bucket of garbled circuits and there is a $\log n$-depth tree of such buckets. The number of garbled circuits that reside within each bucket varies, buckets in lower levels has less garbled circuits, the total number of garbled circuits used as memory entries are $O(T)$. In addition, it has the term $\mathsf{poly}(\kappa, p)$ since it is based on the BMR protocol in which each bit is represented by a κp-bits string. The memory size in our first construction (based on IBE) depends only on the original memory size n, the ORAM's overhead $\log^b n$ and the security parameter κ, hence, it has the simplest expression among the three constructions. The memory size in our second construction (Sec. 5) has the term $O(n \log^{b+1} n)$ since it is based on [15] which adds a tree of keys on top of the ORAM underlying memory size. Note that in both of our constructions the memory size is independent of s (the cut-and-choose parameter). Both [13] and our second construction imply communication and computation complexity proportional to $O(T \log^{a+1} n)$, as they are respectively based on [14] and [15], which incur an overhead of $O(\log n)$ over the ORAM's complexity. Finally, since [14] treats the memory as garbled circuits, it add $O(T)$ garbled circuits over the $O(T \log^a n)$ ones of the ORAM. In another work [38], Miao demonstrates how to achieve persistent data for the two-party setting in the programmable random oracle model, using techniques from [40] and [4], where the underlying one-way function is used in a black-box manner.

	Assump.	BB	Memory	Comm/Comp	No. of GC	Mem. dup.
[13]	OWF	✓	$O(n \log^b n + T)\mathsf{poly}(\kappa, p)$	$O(T \log^{a+1} n)\mathsf{poly}(\kappa, p)$	$O(T \log^a n)$	p
[Sec. 4]	IBE	✗	$O(n \log^b n)\mathsf{poly}(\kappa)$	$O(sT \log^a n)\mathsf{poly}(\kappa)$	$O(sT \log^a n)$	1
[Sec. 5]	OWF	✗	$O(n \log^{b+1} n)\mathsf{poly}(\kappa)$	$O(sT \log^{a+1} n)\mathsf{poly}(\kappa)$	$O(sT \log^{a+1} n)$	1

Table 1: A comparison between the constructions presented in [13] and in this paper. *Memory* refers to the memory size that is needed to be stored by the parties when the original memory size is $|D| = n$; *Comm/Comp* counts the communication and computation complexities of the constructions when the program's original runtime is T; *No. of GC* counts the number of garbled circuits that have to be generated in the protocol; Finally *Mem. dup.* refers to the required number of memory duplications.

2 Preliminaries

Basic Notations. We denote the computational and statistical security parameters by κ and s respectively. 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 $[d] \cup \{0\}$.

We now specify the definition of (κ, s)-computational indistinguishability (denoted $\overset{\kappa,s}{\approx}$), while the usual (computational indistinguishability) definition (denoted $\overset{\text{c}}{\approx}$) 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 $\mathcal{D}$, every $s \in \mathbb{N}$ every positive polynomial $p(\cdot)$ and all sufficiently large κ it holds that for every $a \in \{0,1\}^*$:*

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

2.1 Garbled Circuits

A garbled circuit $(\mathsf{Garb}, \mathsf{Eval})$ [50] is a cryptographic non-interactive object that supports correctness and privacy. In more details, a sender uses Garb to encode a Boolean circuit, that computes some polynomial time function f, in a way that (computationally) hides from the receiver any information about f except for its output. Where the receiver extracts the output using algorithm Eval. In this work we combine the notions of *garbled circuits* and the *cut-and-choose* technique in order to support a malicious sender. Specifically, the sender uses the algorithm Garb to generate s garbled versions $\{\tilde{\mathrm{C}}_i\}_{i\in[s]}$ of a circuit C and some statistical parameter s, as well as their corresponding encoded inputs $\{\tilde{x}_i\}_{i\in[s]}$. The evaluator then chooses a subset $Z \subset [s]$ and uses Eval to evaluate the garbled circuits from this set. Upon completing the evaluation, the evaluator learns $|Z|$ sets of output-wire labels $\{\tilde{y}_i\}_{i\in Z}$ from which it outputs the majority.[1] In the following exposition we use the notation of $\mathsf{lbl}^{j,i}_{\mathrm{in},b}$ to denote the jth input label of the bit-value $b \in \{0,1\}$ for the ith garbled circuit. Analogously, $\mathsf{lbl}^{j,i}_{\mathrm{out},b}$ represents the same notation corresponding to an output wire.

We further abstract two important properties of *authenticity* and *input consistency*. Loosely speaking, authenticity 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 input 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 $\mathcal{F}_{\mathrm{IC}}$ and formally described in Figure 6. That is, given a set of garbled circuits $\{\tilde{\mathrm{C}}_i\}_i$ 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.

We next proceed with our formal definition of garbled circuits.

Definition 2.2 (Garbled circuits.)**.** *A circuit garbling scheme with wire labels consists of the following two polynomial-time algorithms:*

- *The garbling algorithm* Garb*:*

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

  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 $\{\tilde{\mathrm{C}}_i\}_{i\in[s]}$ and s sets of input labels $\{u, b, \mathsf{lbl}^{u,i}_{\mathrm{in},b}\}_{u,i,b}$.

[1]The cut-and-choose analysis ensures that, with overwhelming probability, the majority of these evaluations will correspond to the correct output of C, condition on the remaining garbled circuits from $[s]/Z$ being correctly formed.

- *The evaluation algorithm* Eval*:*

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

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

Furthermore, the following properties hold.

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

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

where $\left(\{\tilde{\mathrm{C}}_i\}_i,\{u,b,\mathsf{lbl}_{\text{in},b}^{u,i}\}_{u,i,b}\right)\leftarrow\mathsf{Garb}\left(1^\kappa,s,\mathrm{C},\{v,b,\mathsf{lbl}_{\text{out},b}^{v,i}\}_{v,i,b}\right)$ 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 $\left\{\mathsf{lbl}_{\text{out}}^{1,z},\ldots,\mathsf{lbl}_{\text{out}}^{v_{\text{out}},z}\right\}_{z\notin[Z]},\{v,b,\mathsf{lbl}_{\text{out},b}^{v,z}\}_{v,z\in[Z],b}$ (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

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

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

Authenticity. We describe the authenticity game in Figure 7 (Appendix A.1) where the adversary obtains 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 $\Pr[\mathrm{Auth}_{\mathcal{A}}(1^\kappa,s,\mathrm{C})=1]$ is negligible. Our definition is inspired by the definition from [4] 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 $\mathcal{F}_{\text{IC}}$ functionality described in Figure 6 in Appendix A.1.

Realizations of our garbled circuits notion. We require the existence of a protocol Π_{IC} that securely realizes the functionality $\mathcal{F}_{\text{IC}}$ described in Figure 6, in the presence of malicious adversaries. In Appendix B we exemplify this realization with [35].

2.2 The RAM Model of Computation

We follow the notation from [18] verbatim. We consider a program P that has a random-access to a memory D of size n, which is initially empty. In addition, the program is given 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 the memory throughout the execution. Gentry et al. 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 $(y_1, \dots, y_c) = (P_1(x_1), \dots, P_\ell(x_c))^D$ 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 $1 \le t \le T$ is represented by a circuit that computes the following functionality:

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

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 $i_t^{\mathsf{write}} \in [n] \cap \perp$ (where $\perp$ means "write nothing") and a bit b_t^{write} to write into that location. The computation $P^D(x)$ starts with an initial state $\mathsf{state}_1 = (x_1, x_2)$, corresponding to the parties' "short input" where the initial read bit b_1^{read} is set to 0 by convention. In each step t, the computation proceeds by running $\mathrm{C}^P_{\mathrm{CPU}}(\mathsf{state}_t, b_t^{\mathsf{read}}) = (\mathsf{state}_{t+1}, i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}})$. Namely, we first read from the requested location i_t^{read} by setting $b_{t+1}^{\mathsf{read}} := D[i_t^{\mathsf{read}}]$ and, if $i_t^{\mathsf{write}} \neq \perp$ we write to the specified 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 $\mathrm{C}^P_{\mathrm{CPU}}$ always set $i_t^{\mathsf{write}} = \perp$.

2.2.1 Predictably Timed Writes

The Predictably Timed Writes property (denoted by "ptWrites" in [18]) implies that it is easy to figure out the time t' in which some location was most recently written to given only the current state of the computation and without reading any other values in memory. More formally,

Definition 2.3. *A program execution $P^D(x_1, x_2)$ has* predictably timed writes *if there exists a polynomial size circuit, denoted* $\mathsf{WriteTime}$*, such that for every $t \in [T]$, $t' = \mathsf{WriteTime}(t, \mathsf{state}_t, i_t^{\mathsf{read}})$ is the largest time (where $t' < t$) in which memory location i_t^{read} has been written; i.e.* $\mathsf{WriteTime}(t, \mathsf{state}_t, i_t^{\mathsf{read}}) = max\left\{t' \mid t' < t \wedge i_{t'}^{\mathsf{write}} = i_t^{\mathsf{read}}\right\}$.

In [18, Appendix A.3] the authors describe how to transform a program without ptWrites into a program with ptWrite, which incurs overhead of $O(\log n)$ in memory access time. The authors further prove the following theorem [18, Theorem D.1]:

Theorem 2.4. *If G is a garbled RAM scheme that provides UMA security and supports programs with ptWrites and O is an ORAM with ptWrites then there exists a garbled RAM scheme G' with full security supporting arbitrary programs.*

Theorem 2.8 extends this theorem to the malicious setting.

2.3 Oblivious RAM (ORAM)

ORAM, initially proposed by Goldreich and Ostrovsky [19, 41, 22], is an approach for making the access pattern of a RAM program input-oblivious. More precisely, it allows a client to store private data on an untrusted server and provides oblivious access to data, by locally storing only a short local state. A secure ORAM scheme not only hides the content of the memory from the server, but also the access pattern, i.e. which locations in memory the client is reading/writing. 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 entirety. 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 work of Goldreich and Ostrovsky [22], ORAM has been extensively studied [45, 23, 31, 49, 46, 44, 52, 8], optimizing different metrics and parameters.

We denote the sequence of memory indices and data written to them in the course of the execution of a program P by $\mathsf{MemAccess}(P, n, x) = \{(i_t^{\mathsf{read}}, i_t^{\mathsf{write}})\}_{t\in[T]}$ and the number of P's memory accesses by $T(P, n, x)$ (i.e. P's running time over memory size n and input x). We define an Oblivious RAM as follows (the definition is the same as in [9]).

Definition 2.5. *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 $x \in \{0,1\}^*$ it follows that $T(P^*(n, x)) \le c(n) \cdot T(P, n, x)$ and there exists a negligible function μ such that the following properties hold:*

- ***Correctness.*** *For any $n \in \mathbb{N}$, any input $x \in \{0,1\}^*$ with probability at least $1 - \mu(\kappa)$, $P^*(n, x) = P(n, x)$.*
- ***Obliviousness.*** *For any two programs P_1, P_2, any $n \in \mathbb{N}$ and any two inputs $x_1, x_2 \in \{0,1\}^*$ if $T(P_1(n, x_1)) = T(P_2(n, x_2))$ and $P_1^* \leftarrow C(n, P_1)$, $P_2^* \leftarrow C(n, P_2)$ then $\mathsf{MemAccess}(P_1^*(n, x_1))$ and $\mathsf{MemAccess}(P_2^*(n, x_2))$ are computationally indistinguishable.*

Note that the above definition (just as the definition from [22]) only requires an oblivious compilation of deterministic programs P. This is without loss of generality as we can always view a randomized program as a deterministic one that receives random coins as part of its input.

2.4 Secure Computation in the RAM Model

We adapt the standard definition for secure two-party computation of [20, 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 an initially empty memory of size n. The running time of the program, denoted T, is bounded by a polynomial in the inputs lengths. Using the notations from Section 2.2, we refer to this (potentially random) process by $P^D(x_1, x_2)$. In this work we prove the security in the presence of malicious computationally bounded adversaries.

We next formalize the ideal and real executions, considering D as a common resource.[2] 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 the 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 $P^D(x_1, x_2)$. We present here both definitions, starting with the definition of full security.

2.4.1 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 $i \in \{\mathrm{S}, \mathrm{R}\}$ 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.

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

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

Input: Upon receiving input value $(\textsc{Input}_{\mathrm{S}}, x_1)$ from S and input value $(\textsc{Input}_{\mathrm{R}}, x_2)$ 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 $(\textsc{Output}_{\mathrm{R}}, T, y)$ to R where T is the running time of $P^D(x_1, x_2)$.

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

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 $i \in \{\mathrm{S}, \mathrm{R}\}$ 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.

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.

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

Definition 2.6 (Secure computation)**.** *Let $\mathcal{F}_{\mathrm{RAM}}$ 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 $i \in \{\mathrm{S}, \mathrm{R}\}$,*

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

where s and κ are the security parameters.

A note on Definition 2.6. Note that in the RAM model the input may be very small while the memory may be very large. Even though, we are restrained from allowing $n = |D|$ be exponential in $|x| = |x_1| + |x_2|$ since, yet, $|x|$ may be larger than $\kappa + s$ and thus means We intentionally set $n = |D| = \mathsf{poly}(\kappa, s)$ and explicitly exclude the case

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.

2.4.2 The UMA Model

In [18], 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. 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. 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 $\mathcal{F}_{\mathrm{UMA}}$. We define security in the UMA model and then discuss our general transformation from UMA to full security.

Definition 2.7 (Secure computation in the UMA model)**.** *Let $\mathcal{F}_{\mathrm{UMA}}$ 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 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*

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

where s and κ are the security parameters.

2.4.3 From UMA to Full Security

Given a UMA-secure protocol for RAM programs that support ptWrites (Definition 2.3) and an ORAM scheme, in [18] the authors presented a way to achieve a fully-secure protocol. Their result, adapted to the malicious setting follows:

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

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

Input: Upon receiving input value $(\textsc{Input}_{\mathrm{S}}, x_1)$ from S and input value $(\textsc{Input}_{\mathrm{R}}, x_2)$ 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 $(\textsc{Output}_{\mathrm{R}}, T, y, \mathsf{MemAccess})$ to R, where T is the running time of $P^D(x_1, x_2)$ and $\mathsf{MemAccess}$ is the access pattern of the execution.

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

Theorem 2.8. *[18, Theorem D.1] Let π be a secure two-party protocol that provides UMA security against a malicious adversary for RAM programs that support ptWrites in the presence of malicious adversaries and an ORAM compiler, denoted C, then there exists a transformation Θ that is given π and C and outputs a fully-secure protocol π'.*

Informally, their transformation requires the party to first run the ORAM algorithms for the initialization of the memory D and compiling the program P in a secure computation to obtain the oblivious memory D^* and oblivious program P^* and then run π over P^* and D^*. The first step provides obliviousness while the second step provides secure memory accesses (privacy and authenticity).

2.4.4 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 (variants of) Merkle trees [14]. In [18] 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.

2.5 Timed IBE [18]

TIBE was introduced by Gentry et al. in [18] 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 [6, 5]. 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:[3]

Definition 2.9 (Timed IBE (TIBE)). *A TIBE scheme Consists of* 5 PPT *algorithms* $\mathsf{MasterGen}, \mathsf{TimeGen}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}$ *with the syntax:*

- $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MasterGen}(1^\kappa)$*: generates master public/secret key pair* $\mathsf{MPK}, \mathsf{MSK}$.
- $\mathsf{TSK}_t \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, t)$*: 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:

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

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

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

2.6 Garbled RAM Based on IBE [18]

Our starting point is the garbled RAM construction of [18]. Intuitively speaking, garbled RAM [37] is an analogue object of garbled circuits [50, 4] 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 [18], the programs can only be executed in the specified order, where each program obtains

[3]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.

a state that depends on prior executions. The [18] garbled RAM proposes a fix to the aforementioned circularity issue raised in [37] by using an Identity Based Encryption (IBE) scheme [6, 5] instead of a symmetric-key encryption scheme.

In more details, the inputs D, P, x to the garbled RAM are garbled into $\widetilde{D}, \widetilde{P}, \widetilde{x}$ such that the evaluator reveals the output $\widetilde{P}(\widetilde{D}, \widetilde{x}) = P(D, x)$ 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 $\widetilde{P}$), 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 [37] 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 [18], the proof of [37] 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 $\widetilde{D}$ 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 $\mathrm{GC}^t_{\mathrm{CPU}}$ is hardwired with the master public key MPK of an IBE scheme.[4] 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, $\mathsf{ct}_b = \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id} = (i, b); \mathsf{msg} = \mathsf{lbl}^{t+1}_b)$ and the correct label is extracted by decrypting the right ciphertext using $\mathsf{sk}_{i,b}$, such that $\mathsf{lbl}^{t+1}_0, \mathsf{lbl}^{t+1}_1$ 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 $t' = \mathsf{WriteTime}(t, \mathsf{state}_t, i^{\mathsf{read}}_t)$. Namely, if at time step t the garbled circuit $\mathrm{GC}^t_{\mathrm{CPU}}$ instructs to read from location i^{read}_t, then the circuit further computes the last time step, u, in which that i^{read}_t was written to, it then computes the translation map $\mathsf{translate}_t = (\mathsf{ct}_0, \mathsf{ct}_1)$ by $\mathsf{ct}_b = \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id} = (u, i^{\mathsf{read}}_t, b); \mathsf{msg} = \mathsf{lbl}^{t+1}_b))$, and outputs it in the clear.

[4]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 2.5 for more details.

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. [18] 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 $(\mathsf{MPK}, \mathsf{MSK})$ whereas the second level has T timed secret keys $\mathsf{TSK}_1, \ldots, \mathsf{TSK}_T$. The keys $\mathsf{MPK}, \mathsf{MSK}$ 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.

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 is applicable for every garbling scheme in the sense of definition 2.2. Recall first that in [18] 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 3 and a high-level overview of our protocol can be found in Section 4.

3.1 Enhanced CPU-Step Function

The enhanced $\mathsf{cpustep}^+$ function is essentially the CPU-step functionality specified in Section 2.2 enhanced with more additional inputs and output, and defined as follows

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

where the additional inputs $\mathsf{MSK}, \mathsf{TSK}_t$ 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 $\mathsf{translate}_t$ is a pair of ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$, encrypted under MPK, that allows the evaluator to obtain the appropriate label of the wire that corresponds

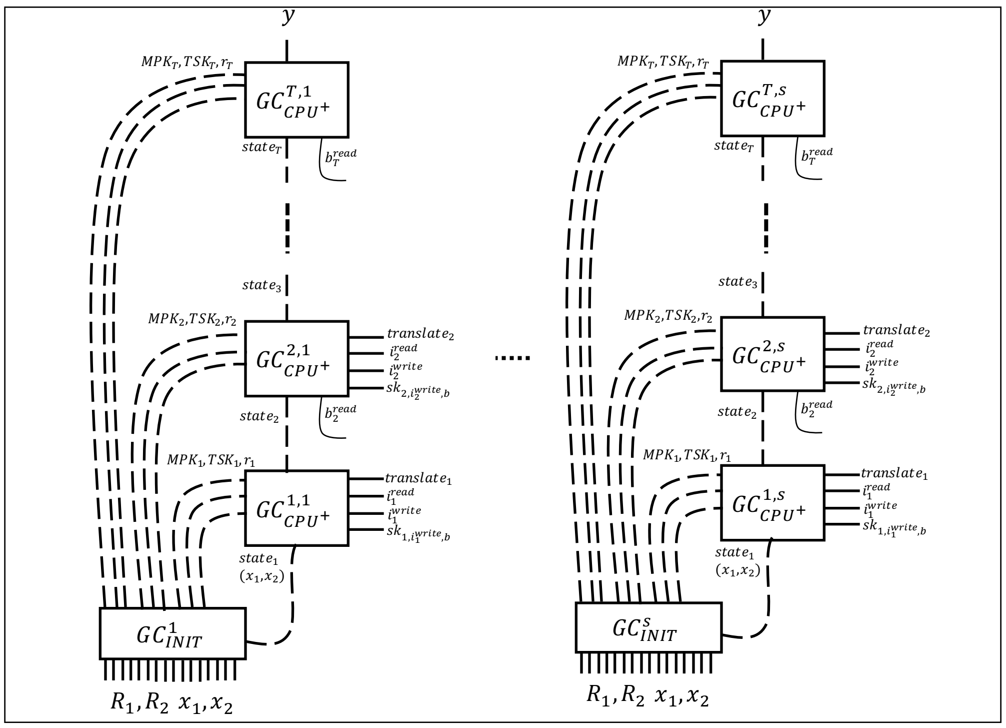

Figure 3: Garbled chains $\mathrm{GC}_{\mathrm{INIT}}, \mathrm{GC}^{1,i}_{\mathrm{CPU}^+}, \ldots \mathrm{GC}^{T,i}_{\mathrm{CPU}^+}$ for $i \in [s]$. Dashed lines refer to values that are passed privately (as one label per wire) whereas solid lines refer to values that are given in the clear.

to the input bit in the next circuit. We denote the circuit that computes that function by $\mathrm{C}^t_{\mathrm{CPU}^+}$. The functionality of $\mathrm{C}^t_{\mathrm{CPU}^+}$ is described in Figure 4. 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 2.3.

3.2 Initialization Circuit

The initialization circuit generates all required keys and randomness to our solution and securely transfers 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 5.

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 [35]. See Figure 8 (Appendix A.3) for its formal definition.

In addition to the above, our protocol uses the following building blocks: A garbling scheme $\pi_{\mathrm{GC}} =$

Enhanced CPU-Step Circuit $\mathrm{C}^t_{\mathrm{CPU}^+}$

This circuit computes the enhanced CPU-step function $\mathsf{cpustep}^+$. This circuit wraps the following algorithms: (1) the usual cpu-step for computing the next CPU-step of program P, (2) $\mathsf{WriteTime}$ which computes the last time t' that the program wrote to location i^{read}_t and (3) the TIBE related functionalities KeyGen and Enc. Furthermore, the labels lbl^{t+1}_0 and lbl^{t+1}_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^{read}_0 to be zero.
- b^{read}_t - the last bit that was read from the memory data (i.e. b^{read}_t was read from location i^{read}_t).
- 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 $\mathsf{Enc}_{\mathsf{MPK}}$.

Outputs. $\mathrm{C}^t_{\mathrm{CPU}^+}$ invokes $\mathrm{C}^t_{\mathrm{CPU}}$ (the usual CPU-step circuit) that computes:

$$\mathsf{cpu\text{-}step}(\mathsf{state}_t, b^{\mathsf{read}}_t) = (\mathsf{state}_{t+1}, i^{\mathsf{read}}_t, i^{\mathsf{write}}_t, b^{\mathsf{write}}_t)$$

where state_{t+1} is the next state of the program; i^{read}_t is the next location to read from; i^{write}_t is the next location to write to and b^{read}_t is the bit to write to location i^{write}_t.
The circuit outputs the translation $\mathsf{translate}_t = (\mathsf{ct}^0_t, \mathsf{ct}^1_t)$ defined by:

$$\begin{aligned} t' &= \mathsf{WriteTime}(t, \mathsf{state}_t, i^{\mathsf{read}}_t) \\ \mathsf{ct}^0_t &= \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id} = (t, t', 0), \mathsf{msg} = \mathsf{lbl}^0_{t+1}) \\ \mathsf{ct}^1_t &= \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id} = (t, t', 1), \mathsf{msg} = \mathsf{lbl}^1_{t+1}) \end{aligned}$$

Finally, the circuit computes $\mathsf{sk}_{(t,i,b)} = \mathsf{KeyGen}(\mathsf{TSK}_t, \mathsf{id} = (t, i^{\mathsf{write}}_t, b^{\mathsf{write}}_t))$ and outputs

$$(\mathsf{state}_{t+1}, i^{\mathsf{read}}_t, i^{\mathsf{write}}_t, \mathsf{sk}_{(t,i,b)}, \mathsf{translate}_t).$$

Figure 4: The CPU-step circuit.

$(\mathsf{Garb}, \mathsf{Eval})$ that preserves the security properties from Definition 2.2; Timed IBE scheme (Section 2.5) $\pi_{\mathrm{TIBE}} = (\mathsf{MasterGen}, \mathsf{TimeGen}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ with security as specified in Definition 2.9 and a statistically binding commitment scheme Com.

4 Constat Round Malicious 2PC for RAM Programs

Given the building blocks detailed in Section 3, we are now ready to introduce our complete protocol. Our description incorporates ideas from both [35] and [18]. Specifically, we borrow the cut-and-choose technique and the cut-and-choose OT abstraction from [35] (where the latter tool enables to ensure input consistency for the *receiver*). Moreover, we extend the garbled RAM ideas presented in [18] for a maliciously

Initialization Circuit C_{INIT}

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

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: $\mathsf{MasterGen}$, $\mathsf{TimeGen}$, 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 $\mathrm{C}^t_{\text{CPU}^+}$ encrypts the two labels of the input wire that corresponds to the input bit of the next circuit $\mathrm{C}^{t+1}_{\text{CPU}^+}$.)

Then, the circuit computes:

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

Outputs.

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

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

Figure 5: Initialization Circuit C_{INIT}.

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.

- **Chains.** Considering a sequence of circuits $\mathrm{C}_{\text{INIT}}, \mathrm{C}^1_{\text{CPU}^+}, \ldots, \mathrm{C}^T_{\text{CPU}^+}$ as a *connected chain of circuits*, the sender S first generates s versions of garbled chains $\mathrm{GC}^i_{\text{INIT}}, \mathrm{GC}^{1,i}_{\text{CPU}^+}, \ldots, \mathrm{GC}^{T,i}_{\text{CPU}^+}$ 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 $\mathsf{lbl}_{\text{out}}$ are chosen uniformly and are fed, together with the circuit $\mathrm{C}^T_{\text{CPU}^+}$, to procedure Garb, which in turn, outputs s sets of input labels. This process is being repeated until 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}.

- **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 Section 2.1.

- **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 2.2, verifying the correctness of the check chains can be done given only (both of the) input labels for $\mathrm{C}_{\mathrm{INIT}}$ circuits.

Achieving full security. In the next step we apply the general transformation discussed in Section 2.4, from UMA to full security.

4.1 2PC in the UMA Model

We proceed with the formal detailed description of our protocol.

Protocol Π^{P}_{UMA} 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 $\mathrm{C}_{\mathrm{INIT}}, \mathrm{C}^1_{\mathrm{CPU^+}}, \ldots, \mathrm{C}^T_{\mathrm{CPU^+}}$ (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 $\mathrm{C}^t_{\text{CPU}^+}$ by calling

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

   for $v \in [v_{\text{out}}], i \in [s], b \in \{0,1\}$ and r^t_g the randomness used within Garb.
   - Interpret the result labels $\{\mathsf{lbl}^{u,i,t}_{\text{in},b}\}_{u,i,b}$ as the following groups of values: state_t, b^{read}_t, MPK_t, TSK_t and r_t, that cover the labels: $\{\mathsf{lbl}^{u,i,t}_{\mathsf{state},b}\}_{u,i,b}$, $\{\mathsf{lbl}^{u,i,t}_{b^{\mathsf{read}}_t,b}\}_{u,i,b}$, $\{\mathsf{lbl}^{u,i,t}_{\mathsf{MPK}_t,b}\}_{u,i,b}$, $\{\mathsf{lbl}^{u,i,t}_{\mathsf{TSK}_t,b}\}_{u,i,b}$, $\{\mathsf{lbl}^{u,i,t}_{r_t,b}\}_{u,i,b}$ resp.

   (b) Garble the remaining CPU-step circuits. For $t = T-1, \ldots, 1$:
   - Hard-code the labels $\{\mathsf{lbl}^{u,i}_{b^{\mathsf{read}}_{t+1},b}\}_{u,i,b}$ inside $\mathrm{C}^t_{\text{CPU}^+}$.
   - Choose random labels for the output wires that correspond to i^{read}_t, i^{write}_t, $\mathsf{sk}_{t,i,b}$ and $\mathsf{translate}_t$ and unite them with the labels $\{\mathsf{lbl}^{u,i,t+1}_{\mathsf{state},b}\}_{u,i,b}$ correspond to state_{t+1} obtained from the previous invocation of Garb; denote the resulting set $\{\mathsf{lbl}^{v,i,t}_{\text{out},b}\}_{v,i,b}$.
   - Garble $\mathrm{C}^t_{\text{CPU}^+}$ by calling

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

   with $\{\mathsf{lbl}^{v,i,t}_{\text{out},b}\}_{v,i,b}$ the set of labels from above and r^t_g the randomness used within Garb.
   - Interpret the result labels $\{\mathsf{lbl}^{u,i,t}_{\text{in},b}\}_{u,i,b}$ as the following groups of values: state_t, b^{read}_t, MPK_t, TSK_t and r_t, that cover the labels: $\{\mathsf{lbl}^{u,i,t}_{\mathsf{state},b}\}_{u,i,b}$, $\{\mathsf{lbl}^{u,i,t}_{b^{\mathsf{read}}_t,b}\}_{u,i,b}$, $\{\mathsf{lbl}^{u,i,t}_{\mathsf{MPK}_t,b}\}_{u,i,b}$, $\{\mathsf{lbl}^{u,i,t}_{\mathsf{TSK}_t,b}\}_{u,i,b}$, $\{\mathsf{lbl}^{u,i,t}_{r_t,b}\}_{u,i,b}$ resp.

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

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

- Interpret the input labels result from that invocation of Garb by $\{\mathsf{lbl}^{u,i}_{\mathrm{S},b}\}_{u,i,b}$ and $\{\mathsf{lbl}^{u,i}_{\mathrm{R},b}\}_{u,i,b}$ 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 $\mathbf{v_1}, \dots, \mathbf{v}_\ell$ 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 $\mathrm{GC}^1_{\mathrm{INIT}}, \dots, \mathrm{GC}^s_{\mathrm{INIT}}$.

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

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

3. SEND GARBLED CIRCUITS AND COMMITMENTS.

   S sends R the garbled circuits chains $\mathrm{GC}^i_{\mathrm{INIT}}, \mathrm{GC}^{1,i}_{\mathrm{CPU+}}, \dots, \mathrm{GC}^{T,i}_{\mathrm{CPU+}}$ for every $i \in [s]$, and the commitment $\mathsf{com}^{u,i}_b = \mathsf{Com}(\mathsf{lbl}^{u,i}_{\mathrm{S},b}, \mathsf{dec}^{u,i}_b)$ for every label in $\{\mathsf{lbl}^{u,i}_{\mathrm{S},b}\}_{u,i,b}$ where $\mathsf{lbl}^{u,i}_{\mathrm{S},b}$ is the bth label ($b \in \{0,1\}$) for the sender's uth bit ($u \in [\ell]$) for the ith garbled circuit $\mathrm{GC}_{\mathrm{INIT}}$.

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 $\mathrm{GC}^z_{\mathrm{INIT}}$ for every $z \in Z$. If the values received by S are incorrect, it outputs $\perp$ and aborts. Chains $\mathrm{GC}^z_{\mathrm{INIT}}, \mathrm{GC}^{1,z}_{\mathrm{CPU+}}, \dots, \mathrm{GC}^{t,z}_{\mathrm{CPU+}}$ 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 $\mathrm{GC}^z_{\mathrm{INIT}}$ (from Steps 2 and 5). This means that it can check the correctness of the chains $\mathrm{GC}^z_{\mathrm{INIT}}, \mathrm{GC}^{1,z}_{\mathrm{CPU+}}, \dots, \mathrm{GC}^{T,z}_{\mathrm{CPU+}}$ 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 $\left\{(\mathsf{lbl}^{1,z}_{\mathrm{in},x_1[1]}, \dots, \mathsf{lbl}^{\ell,z}_{\mathrm{in},x_1[\ell]})\right\}_{z \notin [Z]}$ 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 $\mathrm{C}_{\mathrm{INIT}}$, its garbled versions $\{\mathrm{GC}^i_{\mathrm{INIT}}\}_{z \notin Z}$ and the labels $\left\{(\mathsf{lbl}^{1,z}_{\mathrm{in},x_1[1]}, \dots, \mathsf{lbl}^{\ell,z}_{\mathrm{in},x_1[\ell]})\right\}_{z \notin [Z]}$ that were sent before.
     - S inputs its randomness r^0_g and the set of output labels $\{\mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b}$ that were used within Garb on input $\mathrm{GC}_{\mathrm{INIT}}$, along with the decommitments $\{\mathsf{dec}^{u,z}_b\}_{u \in [\ell], z \notin Z, b \in \{0,1\}}$.

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 $\mathrm{GC}^{z}_{\mathrm{INIT}}$ using Eval and the input wires it obtained in Step 7 and reveal one label for each of its output wires $\mathsf{lbl}^{\mathrm{out},z}_{\mathrm{INIT}}$.

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

      i. For every $z \in \tilde{Z}$, evaluate $\mathrm{GC}^{t,z}_{\mathrm{CPU}^+}$ using Eval and obtain one output label for each of its output wires, namely, $\mathsf{lbl}^{\mathrm{out},t,z}_{\mathrm{CPU}^+}$. Part of these labels refer to $\mathsf{state}_{t+1,z}$. In addition Eval outputs $\mathsf{out}_{t,z} = (i^{\mathsf{read}}_{t,z}, i^{\mathsf{write}}_{t,z}, b^{\mathsf{write}}_{t,z}, \mathsf{translate}_{t,z})$ in the clear[5]. For $t = T$ Eval outputs state_{T+1} in the clear and we assign $\mathsf{out}_{t,z} = \mathsf{state}_{T+1,z}$.

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

      iii. Output out_{T+1}.

We prove the following theorem (for further details about the hybrid model see A.2).

Theorem 4.1. *Assume π_{GC} is a garbling scheme (cf. Definition 2.2), π_{TIBE} is TIBE scheme (cf. Definition 2.9) and* 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 (cf. Definition 2.3).*

- *Let $n = |D|$ be the size of the storage, T be the program's run-time and $|x| = |x_1| + |x_2|$ be the overall input length to $\mathcal{F}_{\mathrm{UMA}}$. Then, in protocol Π^{P}_{UMA} the size of the garbled storage is $O(n \cdot \kappa)$, the size of the garbled input is $|x| \cdot O(\kappa)$ and the size of the garbled program and its evaluation time are $O(T \cdot s \cdot \kappa)$.*

- *In addition, there exist a secure protocol that realizes $\mathcal{F}_{\mathrm{RAM}}$ with garbled storage of size $n \cdot \mathsf{poly}(\kappa, \log n)$, garbled input of size $|x| \cdot O(\kappa)$ and garbled program and evaluation time of $T \cdot \mathsf{poly}(\kappa, \log n, s)$.*

Proof: We begin our proof by analyzing the garbled storage, program and input complexities; the security proof is in Section 4.2.

Note that the size of the initialization circuit $\mathrm{C}_{\mathrm{INIT}}$ is $O(T \cdot \kappa)$ where the bound on the randomness used by the IBE algorithms is $O(\kappa)$. This is because the circuit evaluates the IBE algorithms $O(T)$ times, each such sub-circuit is of size $O(\kappa)$. In addition, all components inside the enhanced CPU-step circuit $\mathrm{C}^{t}_{\mathrm{CPU}^+}$ are of size $O(\kappa)$. Since the sender garbles s chains (where s is a statistical security parameter) the overall number of garbled circuits is $O(T \cdot s)$ and their total size is $O(T \cdot s \cdot \kappa)$. It is clear that the communication and computation complexities depend on the number of garbled circuits and their total size. In particular, our protocol requires $O(T \cdot s)$ oblivious transfers followed by sending and evaluating $O(T \cdot s)$ garbled circuits of total size $O(T \cdot s \cdot \kappa)$, which leads to communication and computation complexities of $O(T \cdot s \cdot \kappa)$. Finally, for each bit in the memory the evaluator stores $O(\kappa)$ bits. Recalling that using the cut-and-choose with our technique of factoring out the randomness to the initialization circuit implies that the receiver stores a *single* copy of the memory. Thus, the memory size is $O(n \cdot \kappa)$.

In transformation Θ (implied by Theorem 2.8) the parties apply an ORAM compiler to the original program, this means that the runtime of the result program increases by a factor of polylog n (where n is the

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

memory size). It holds that the overall number of garbled circuits is $T \cdot \mathsf{poly}(s, \log n)$ and their total size is $T \cdot \mathsf{poly}(s, \kappa, \log n)$. Similarly, since there are $\mathsf{polylog}\ n$ more garbled circuits, the communication and computation complexities are now $T \cdot \mathsf{poly}(s, \kappa, \log n)$ and the memory size is $n \cdot \mathsf{poly}(\kappa, \log n)$.

■

4.2 Security Proof of Theorem 4.1

We first show the intuition of how our protocol achieves security in the UMA model whereas a full proof of lemma 4.1 presented at Section 4.2.1. 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 [35], 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 here is 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 $\mathsf{translate}$ 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.

The selective-bit attack in the memory. In the context of secure computation via garbled circuit, the "selective-bit attack" means that the garbler may construct an input gate[6] such that if the evaluator's input on some input wire is b the evaluation proceeds successfully and otherwise, if the input is $1 - b$ then the outcome of that gate is some "gibberish" key that leads to an evaluation failure. This way, the garbler can learn a single input bit of the evaluator, just by inspecting whether it aborts or not.

In our construction, the selective-bit attack is completely thwarted, both in garbled circuits and in memory. That is, the cut-and-choose technique assures that the garbler constructed the garbled circuits correctly (with overwhelming probability). Now, conditioning on the event that all garbled circuits are correct, the only way the garbler affects memory contents is by providing its part of the data for the one-time initialization phase, which takes place before the program is being executed, or by providing its input to the

[6]The attack is not limited to the input gates but it easier to describe this way.

program being executed. Both kinds of behavior is allowed even in the ideal model, thus, it holds that the cut-and-choose technique protect the evaluator from selective-bit attack in memory as well.

4.2.1 A Formal Proof

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 $\mathcal{F}_{\rm CT}$ and the input consistency check functionality $\mathcal{F}_{\rm IC}$. We separately prove the case that $\rm S$ is corrupted and the case that $\rm R$ is corrupted.

The case $\rm S$ is corrupted. This case is very similar to the case in which $\rm S$ is corrupted in a standard cut-and-choose based protocol (e.g. [35]). Intuitively, $\rm S$ 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 $\rm S$ 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 $\Pi_{\rm SCCOT}$, $\rm R$ receives all the keys associated with its own input wires for the check circuits in Z (while $\rm S$ knows nothing about Z). Furthermore, $\rm S$ 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 [35], this event occurs with probability of approximately $\frac{1}{2^{s/4}}$. The sender $\rm S$ further participates in an input consistency protocol $\Pi_{\rm IC}$ which proves to $\rm R$ that all its inputs to the evaluation circuits are consistent.

We now proceed to the formal proof. Let $\mathcal{A}$ be an adversary controlling $\rm S$ in an execution of $\Pi^{P}_{\rm UMA}$ where a trusted party is used to compute the cut-and-choose OT functionality $\mathcal{F}_{\rm SCCOT}$ and the input consistency check functionality $\mathcal{F}_{\rm IC}$. We construct a simulator $\mathcal{S}$ that runs in the ideal model with a trusted party computing $\mathcal{F}^{PD}_{\rm UMA}$. The simulator $\mathcal{S}$ internally invokes $\mathcal{A}$ and simulates the honest $\rm R$ for $\mathcal{A}$ as well as the trusted party computing $\mathcal{F}_{\rm SCCOT}$ and $\mathcal{F}_{\rm IC}$ functionalities. In addition, $\mathcal{S}$ interacts externally with the trusted party computing $\mathcal{F}^{PD}_{\rm UMA}$. $\mathcal{S}$ 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 $\mathcal{F}_{\rm SCCOT}$ functionality. These inputs constitute an $\ell \times s$ matrix of label pairs $\{(\mathsf{lbl}^{1,1}_{\rm R,0}, \mathsf{lbl}^{1,1}_{\rm R,1}), \ldots, (\mathsf{lbl}^{\ell,s}_{\rm R,0}, \mathsf{lbl}^{\ell,s}_{\rm R,1})\}$, where $\mathsf{lbl}^{j,i}_{\rm R,b}$ is the label associated with the jth input wire of the receiver $\rm R$ in the ith garbled version of the circuit $\rm C_{INIT}$. 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 $\mathrm{GC}^{i}_{\rm INIT}, \mathrm{GC}^{1,i}_{\rm CPU^+}, \ldots, \mathrm{GC}^{T,i}_{\rm CPU^+}$ for every $i \in [s]$ and $2\ell s$ commitments $\{\mathsf{com}^{u,i}_{b}\}$ for every label $\mathsf{lbl}^{u,i}_{\rm S,b}$ as described in the protocol (the garbled values associated with the sender's input wires to $\{\rm C_{INIT}\}_i$ 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 $\mathsf{lbl}^{1,z}_{\rm R,0}, \mathsf{lbl}^{1,z}_{\rm R,1}$ (i.e. the two labels for the first input wire of $\rm R$ in every check chain, this proves to $\mathcal{A}$ that this chain is indeed a check chain, otherwise, $\rm R$ 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 $\{\mathsf{lbl}^{u,i}_{\rm S,b}, \mathsf{dec}^{u,i}_{b}\}$ where $\mathsf{lbl}^{u,i}_{\rm S,b}$ is the bth label ($b \in \{0,1\}$) for the sender's uth bit ($u \in [\ell]$) for the ith garbled circuit $\rm GC_{INIT}$ and $\mathsf{dec}^{u,i}_{b}$ is its decommitment information. $\mathcal{S}$ verifies that the decommitment information is correct. If not, $\mathcal{S}$ sends $\perp$ to the trusted party, simulates $\rm R$ aborting and outputs whatever $\mathcal{A}$ outputs.

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

6. $\mathcal{S}$ receives labels $\big\{(\hat{\mathsf{lbl}}^{1,z}_{\mathrm{in},x_1[1]}, \ldots, \hat{\mathsf{lbl}}^{\ell,z}_{\mathrm{in},x_1[\ell]})\big\}_{z \notin [Z]}$. In addition $\mathcal{S}$, as a trusted party in the input consistency protocol Π_{IC}, receives the randomness r^0_g, the output labels $\{\mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b}$ that were used by $\mathcal{A}$ to generate the s garbled chains in Step 1 of the protocol, together with the decommitments $\mathsf{dec}^{u,i}_{b}$ 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 $\perp$, simulates R aborting and outputs whatever $\mathcal{A}$ outputs.
   - Otherwise, for every $u \in [\ell']$ ($|x_1| = \ell'$ as specified above), if $\hat{\mathsf{lbl}}^{u,z}_{\mathrm{in},x_1[u]} = \mathsf{lbl}^{u,z}_{\mathrm{in},0}$ set $x_1[u] = 0$ and if $\hat{\mathsf{lbl}}^{u,z}_{\mathrm{in},x_1[u]} = \mathsf{lbl}^{u,z}_{\mathrm{in},1}$ 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 $(\textsc{Input}_{\mathrm{S}}, x_1)$ to the trusted party computing $\mathcal{F}^{P^D}_{\mathrm{UMA}}$ and outputs whatever $\mathcal{A}$ outputs and halts.

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

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

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 the honest receiver would do and sends S the rest of the values correctly (also, if it caught a cheat, it aborts as a honest receiver would do). Importantly, as 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 $\mathrm{GC}^{i}_{\mathrm{INIT}}, \mathrm{GC}^{1,i}_{\mathrm{CPU}^+}, \ldots, \mathrm{GC}^{T,i}_{\mathrm{CPU}^+}$ 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 $\mathrm{GC}^{i}_{\mathrm{INIT}}, \mathrm{GC}^{1,i}_{\mathrm{CPU}^+}, \ldots, \mathrm{GC}^{T,i}_{\mathrm{CPU}^+}$ 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 $\perp$ to the trusted party whenever R would abort and output $\perp$. This completes the proof of this corruption case.

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 $\mathcal{F}_{\text{SCCOT}}$ functionality. This implies that R can do nothing but honestly evaluate the evaluation circuits, where each final circuit outputs the same value $P^D(x_1, x_2)$. That is, assume that R evaluates the $s/2$ garbled circuits $\mathrm{GC}^{t,z}_{\mathrm{CPU}^+}$ for CPU-step t and all $z \notin Z$; these garbled circuits output a translate $\mathsf{translate}_t$ tuple which corresponds to the values $\mathsf{ct}_0 = \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id}_0, \mathsf{lbl}_0^{t+1})$ and $\mathsf{ct}_1 = \mathsf{Enc}_{\mathsf{MPK}}(\mathsf{id}_1, \mathsf{lbl}_1^{t+1})$.[7] 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^P_{UMA} 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 $\mathcal{F}^{P^D}_{\mathrm{UMA}}$.

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 $\mathcal{F}^{P^D}_{\mathrm{UMA}}$, and halts outputting whatever A outputs.)

2. $\mathcal{S}$ sends $(\textsc{Input}_{\mathrm{R}}, x_2)$ to the trusted party computing $\mathcal{F}^{P^D}_{\mathrm{UMA}}$ and receives the output $(\textsc{Output}_{\mathrm{R}}, T, y)$ and the memory accesses $\mathsf{MemAccess} = \{(i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}})\}_{t\in[T]}$ 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 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 $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MasterGen}(1^\kappa)$ and $\mathsf{TSK}_t \leftarrow \mathsf{TimeGen}(\mathsf{MSK}, t)$ for $t = 1, \ldots, T$.

   (b) For the last time step $t = T$, create $\{\mathrm{GC}^{t,z}_{\mathrm{CPU}^+}\}_z$ by calling SimGC on the circuit $\mathrm{C}^t_{\mathrm{CPU}^+}$ 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 $i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}}$ are given in the clear (from $\mathsf{MemAccess}$). Also, note that the labels $\mathsf{lbl}^{t+1,z}_{\mathsf{read},b}$ for the input bit b of circuit $\mathrm{GC}^{t+1,z}_{\mathrm{CPU}^+}$ had been produced by the simulator in step $t+1$. The simulator $\mathcal{S}$ computes the secret key $\mathsf{sk}_{(t,i,b)}$ and the translation table $\mathsf{translate}_t$ as follows:

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

[7]Recall that all the circuits evaluated in time t output the same value for $\mathsf{translate}_t$ since they all use the same randomness to compute it.

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

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

(d) Generate $\{\mathrm{GC}^{t,z}_{\mathrm{CPU}^+}\}_z$ by calling SimGC on the circuit $\mathrm{C}^t_{\mathrm{CPU}^+}$ such that for the evaluation circuits ($z \notin Z$) it inputs the values $i^{\mathsf{write}}_t, i^{\mathsf{read}}_t, \mathsf{sk}_{(t,i,b)}, \mathsf{translate}_t$ 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 $\{\mathrm{GC}^{1,z}_{\mathrm{CPU}^+}, \ldots, \mathrm{GC}^{T,z}_{\mathrm{CPU}^+}\}_z$ 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 $\{\mathrm{GC}^z_{\mathrm{INIT}}\}_z$. Namely, we have the output labels for x_1, x_2 and $\{\mathsf{MPK}_t, \mathsf{TSK}_t, r^{\mathsf{KeyGen}}_t, r^{\mathsf{Enc}}_{t,0}, r^{\mathsf{Enc}}_{t,1}\}_{t\in[T]}$ (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 $\{\mathrm{C}^z_{\mathrm{INIT}}\}_z$.

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 $\mathrm{GC}^z_{\mathrm{INIT}}$, 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 $\overline{\mathsf{lbl}_Z} = (\mathsf{lbl}^{1,z}_{\mathrm{R},0}, \mathsf{lbl}^{1,z}_{\mathrm{R},1}, \ldots, \mathsf{lbl}^{\ell,z}_{\mathrm{R},0}, \mathsf{lbl}^{\ell,z}_{\mathrm{R},1})$ for all $z \in Z$, and by $\overline{\mathsf{lbl}_{\tilde{Z}}} = (\mathsf{lbl}^{1,z}_{\mathrm{R},x_2[1]}, \ldots, \mathsf{lbl}^{\ell,z}_{\mathrm{R},x_2[\ell]})$ for $z \in \tilde{Z}$. Then, $\mathcal{S}$ hands $\mathcal{A}$ all the above labels, i.e. the union $\overline{\mathsf{lbl}_Z} \cup \overline{\mathsf{lbl}_{\tilde{Z}}}$ 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 $\{\mathrm{GC}^i_{\mathrm{INIT}}\}_{i\in[s]}$.

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 $\perp$ and halts.

   (b) If $Z = Z'$ and any of the values received are incorrect, then $\mathcal{S}$ sends $\perp$ 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 $\mathcal{F}_{\mathrm{IC}}$.

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:

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

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

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 $\mathrm{GC}^{t,z}_{\mathrm{CPU}^+}$ for $z \in Z$. Instead of choosing $\mathsf{translate}_t$ as in the simulation described above, we choose $\mathsf{translate}_t = (\mathsf{ct}_0, \mathsf{ct}_1)$ to both be encryptions of the correct label of the next circuit:

$$\mathsf{ct}_0 \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\mathsf{read}}_t, 0), \mathsf{lbl}^{t+1,z}_{\mathsf{read},0}) \quad , \quad \mathsf{ct}_1 \leftarrow \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\mathsf{read}}_t, 1), \mathsf{lbl}^{t+1,z}_{\mathsf{read},1})$$

where $\mathsf{lbl}^{t+1,z}_{\mathsf{read},0}$ and $\mathsf{lbl}^{t+1,z}_{\mathsf{read},1}$ are the labels corresponding to the bits 0 and 1 for the wire b^{write}_{t+1} in garbled circuit $\mathrm{GC}^{t+1,z}_{\mathrm{CPU}^+}$, 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 $\mathrm{GC}^{z}_{\mathrm{INIT}}$ 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:

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

and

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

We prove this by the following claims:

Claim 4.1. *For each $t \in \{0, \ldots, T\}$ it holds that* $\mathbf{Hyb}_t \overset{\mathrm{c}}{\approx} \mathbf{Hyb}'_{t+1}$.

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 $\mathrm{GC}^{1,z}_{\mathrm{CPU}^+}, \ldots, \mathrm{GC}^{t,z}_{\mathrm{CPU}^+}$ are already simulated and hence they only rely on a subset of the input wire labels for the input $\mathsf{state}_{t+1}, b^{\mathsf{write}}_{t+1}$, 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 $\mathsf{translate}_t$ used to create the tth circuit only encrypts one label and the other ciphertext is "dummy".)

Formally, the difference between the distributions is that the garbled circuits for the $(t+1)$th step in $\mathbf{Hyb}_t$ are generated by Garb whereas in $\mathbf{Hyb}'_{t+1}$ they were are generated by SimGC. In both cases the circuit generates $\mathsf{translate}$ according to the real execution. Therefore, given inputs x_1, x_2 and a distinguisher $\mathcal{D}'$ between these two distributions, we can construct a distinguisher $\mathcal{D}''$ that breaks the privacy of π_{GC}. In particular, the distinguisher $\mathcal{D}''$ is given a set of garbled circuits $\{\mathrm{GC}^{t+1,z}_{\mathrm{CPU}^+}\}_{z \notin Z}$ generated either by the garbling scheme Garb or by its simulator SimGC, where the output wires' labels equal the input wires' labels obtained from garbling $\{\mathrm{GC}^{t+2,z}_{\mathrm{CPU}^+}\}_{z \notin Z}$ (note that by Definition 2.2 the privacy must hold for every choice of such labels). The distinguisher $\mathcal{D}''$ produces garbled circuits $\{\mathrm{GC}^{j,z}_{\mathrm{CPU}^+}\}_{z \notin Z}$ for $j = T, \ldots, t+2$ exactly as in the real execution; then, it plugs in the garbled circuits $\{\mathrm{GC}^{t+1,z}_{\mathrm{CPU}^+}\}_{z \notin Z}$ (received as input) and completes the chain of garbled circuits as in the simulation and outputs the entire chain. Observe that if the garbled circuits $\{\mathrm{GC}^{t+1,z}_{\mathrm{CPU}^+}\}_{z \notin Z}$ (given to $\mathcal{D}''$ as input) are output of Garb then the chain output by $\mathcal{D}''$ is distributed identically to $\mathbf{Hyb}_t$. Otherwise, it is distributed identically to $\mathbf{Hyb}'_{t+1}$. We conclude that the advantage of $\mathcal{D}''$ equals the advantage of $\mathcal{D}'$, which has to be negligible.

■

Claim 4.2. *For each $t \in \{0, \ldots, T\}$ it holds that $\mathbf{Hyb}'_t \overset{c}{\approx} \mathbf{Hyb}_t$.*

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 $\mathsf{translate}_t = (\mathsf{ct}_0, \mathsf{ct}_1)$ used to simulate the tth set of circuits. Let $b = b^{\mathsf{write}}_{t+1}$ be the value of the read-bit in location i^{read}_t in the computation. Then, in $\mathbf{Hyb}'_t$ we set

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

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

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

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 $(t', i^{\mathsf{write}}_{t'}, b^{\mathsf{write}}_{t'})$ 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 $\mathsf{MPK}, \mathsf{TSK}_{t+1}, \ldots, \mathsf{TSK}_T$ from the game along with one secret key for every time step $t' < t$. $\mathcal{D}'$ works as follows:

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

2. For the tth circuits $\mathrm{GC}^{t,z}_{\mathrm{CPU}^+}$, let $b = b^{\mathsf{read}}_t$ 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 $\mathsf{lbl}^{t+1,z}_{\mathsf{read},b}, \mathsf{lbl}^{t+1,z}_{\mathsf{read},1-b}$ 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 $\mathsf{id}^* = (t, i^{\mathsf{write}}_t, b)$ and the two messages: $\mathsf{msg}_0 = \mathsf{lbl}^{t+1,z}_{\mathsf{read},1-b}$ and $\mathsf{msg}_0 = 0$ and receives the ciphertext ct.

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

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

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 $\mathsf{ct} = \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\mathsf{read}}_t, b), \mathsf{lbl}^{t+1,z}_{\mathsf{read},b})$ then the result hybrid is identically distributed to $\mathbf{Hyb}'_t$ and if $\mathsf{ct} = \mathsf{Enc}_{\mathsf{MPK}}((t', i^{\mathsf{read}}_t, b), 0)$ 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 4.3. *It holds that $\mathbf{Hyb}_{(-1)} \overset{c}{\approx} \mathbf{Hyb}_0$.*

Proof: Note that the difference between the hybrids is merely whether the first circuits $\mathrm{C}^z_{\mathrm{INIT}}$ are simulated or not. Hence, we rely on the security of the garbling scheme as done in the proof of Claim 4.2. ■

5 Removing the IBE Assumption

In this section we discuss how to apply our ideas to the GRAM by Garg et al. [15] with the aim of removing the IBE assumption. We begin with briefly describing their scheme and then present our construction.

5.1 Background: GRAM Based on OWF [15]

In this construction, as in the previous GRAM scheme, the garbler first garbles the data D, the program P and the input x, and forwards these to the evaluator that runs the evaluation algorithm to obtain the program output y. More precisely, each internal node node is associated with a PRF key r that is encrypted under a PRF key associated with node's parent, and each memory access is translated into a sequence of $d-1$ navigation circuits (where $d = \log n$ is the depth of the tree) and a step circuit. During the evaluation, each navigation circuit outputs a translation map that allows the evaluator to learn the input labels that encode the input keys associated with the next node on the path to the required memory location. These keys are then used in the next navigation circuit. In addition, the circuit refreshes the PRF key associated with node and computes a new set of PRF values based on this new key to be stored on node. The step circuit finally performs the read or write operation. In more details,

5.1.1 Garbling Data

- Split first the data D into 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 $i \in \{0, \ldots, d-1\}$ is the depth in the tree and j is the index of that node within the ith tree level, where that the ith level includes 2^i keys. For example, when $|D| = 8\kappa$ then $n = 8$ and the nodes on 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.
- Denote the qth bit of a key r and the qth bit of a data block D by r^q and D^q, respectively, where $q \in \{1, \ldots, \kappa\}$. Encrypt each bit of a key 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. Then, the (i,j) node contains $F_{r_{i-1,j/2}}(\mathsf{side}, r^1_{i,j}, 1), \ldots, F_{r_{i-1,j/2}}(\mathsf{side}, r^\kappa_{i,j}, \kappa)$ with $\mathsf{side} = \mathsf{left}$ if (i,j) is the left child of $(i-1, j/2)$ and $\mathsf{side} = \mathsf{right}$ otherwise. For example, the garbled memory in node $(2,1)$ has the bits of key $r_{2,1}$ 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 $r_{1,0}$). Formally we write

$$\widehat{r_{2,1}} = F_{r_{1,0}}(\mathsf{right}, r^1_{2,1}, 1), \ldots, F_{r_{1,0}}(\mathsf{right}, r^\kappa_{2,1}, \kappa) \ and\ \tilde{D}_2 = F_{r_{2,1}}(\mathsf{left}, D^1_2, 1), \ldots, F_{r_{2,1}}(\mathsf{left}, D^\kappa_2, \kappa)$$

- 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 tree is the garbled data $\tilde{D}$. Overall we have that $|\tilde{D}| = O(\kappa \cdot |D|)$.

On a high level, the goal of this data-structure is to prevent a "memory rollback" attack by the evaluator. That is, an attempt to input the CPU-step circuit an outdated value (i.e. value that was in memory once, but got updated already). Using the above machinery, the evaluator does not see the actual keys, and so it cannot modify the ciphertexts in a way that yield a valid evaluation of the next garbled circuit.

5.1.2 Garbling Program

Each step in the original program P is performed using d steps in the new scheme, such that the first $d-1$ steps are used to navigate to the required data block in D_j and the last, dth, 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 $d-1$ navigation steps are carried out by evaluating $d-1$ nav garbled circuits and the last dth step is performed by evaluating a single garbled circuit step. In the following, we overview the internal design of circuits nav and step and further stress that all fresh PRF keys that are hardwired into the circuits in the [15] construction are given as inputs to the circuits in our construction. Below we present the modified version of [15] that is consistent with our modifications.

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 jth circuit in the ith group of circuits. $\mathsf{nav}_{i,j}$ helps to navigate in the jth level of the tree towards the data item D_L that is read in the ith CPU-step. Consider circuit $\mathsf{nav}_{i,j}$:

Inputs:

- **Two (plain) keys** $r_{j+1,\ell}, r_{j+1,\ell+1}$ (where $\ell \in [2^{j+1}]$) from level $j+1$ of the tree (this amounts to 2κ input wires). We stress that the circuit can see these keys but the evaluator sees only wire labels that hide them.
- **The key** $s = r_{j,\ell/2}$. That is, the parent key of $r_{j+1,\ell}, r_{j+1,\ell+1}$. This amounts to κ input wires.
- **A fresh key** $\tilde{r}$ that is going to replace either $r_{j+1,\ell}$ or $r_{j+1,\ell+1}$ in memory. That is, the evaluator is going to store in memory $\{F_s(\mathsf{left}, \tilde{r}^q, q)\}_{q\in[\kappa]}, \{F_s(\mathsf{right}, r^q_{j+1,\ell+1}, q)\}_{q\in[\kappa]}$ or $\{F_s(\mathsf{left}, r^q_{j+1,\ell}, q)\}_{q\in[\kappa]}, \{F_s(\mathsf{right}, \tilde{r}^q, q)\}_{q\in[\kappa]}$). This amounts to κ input wires.
- **The current state** of the program (which travels until the next cpu-step circuit).
- **The index** $L = L_0, \ldots, L_{\log|D|}$ of the destination memory entry to access.

Hardwired values:

- **The value** j, that is, which bit in L the circuit should check in order to decide what to do (see next).
- **A set of** 4κ **input wire labels** of the next circuit (the next circuit is $\mathsf{nav}_{i,j+1}$ if $j+1 < d$ or step_i if $j+1 = d$). Those wire labels correspond to two keys $r_{j+2,m}, r_{j+2,m+1}$ in the $j+2$th level of the memory (or D_m, D_{m+1} if the next circuit is step_i). There are 4κ labels because there are 2 labels (for zero and one) for each bit of those keys. Denote those labels by $\mathsf{label}_0^{\mathsf{left},q}, \mathsf{label}_1^{\mathsf{left},q}, \mathsf{label}_0^{\mathsf{right},q}, \mathsf{label}_1^{\mathsf{right},q}$ for all $q \in [\kappa]$.

Procedure:

- Let $L_j = b$ (this indicates whether we should keep the navigation to the left, if $b = 0$, or to the right, if $b = 1$). Assign $r_{j+1,\ell+b} \leftarrow \tilde{r}$ (i.e. replace the value of $r_{j+1,\ell+b}$), the value of $r_{j+1,\ell+1-b}$ remains the same.

– Perform

$$\widehat{r_{j+1,\ell}} = F_s(\mathsf{left}, r^1_{j+1,\ell}, 1), \ldots, F_s(\mathsf{left}, r^\kappa_{j+1,\ell}, \kappa)$$
$$\widehat{r_{j+1,\ell+1}} = F_s(\mathsf{right}, r^1_{j+1,\ell+1}, 1), \ldots, F_s(\mathsf{right}, r^\kappa_{j+1,\ell+1}, \kappa)$$

– Let $r_{j+2,m}$ and $r_{j+2,m+1}$ be the keys associated with the children of $r_{j+1,\ell+b}$ (i.e. one of these keys is on the path toward data block with index L). Recall that (1) the evaluator knows the PRF values $F_{r_{j+1,\ell+b}}(\mathsf{left}, r^q_{j+2,m}, q)$ and $F_{r_{j+1,\ell+b}}(\mathsf{right}, r^q_{j+2,m+1}, q)$ for all $q \in [\kappa]$ in its garbled memory and (2) the next navigation circuit should receive $r^q_{j+2,m}$ and $r^q_{j+2,m+1}$ as inputs. Also, recall that the input wire labels by which these 2κ bits are transferred to $\mathsf{nav}_{i,j+1}$ are hardwired and known to the current circuit $\mathsf{nav}_{i,j}$. The current circuit $\mathsf{nav}_{i,j}$ does not know the values of the keys $r^q_{j+2,m}$ and $r^q_{j+2,m+1}$ but since they are encrypted bit-by-bit using the key $r_{j+1,\ell+1}$, it is possible to output a translation table to be used by the evaluator, see immediately. Thus, let $\mathsf{label}_0^{\mathsf{left},q}, \mathsf{label}_1^{\mathsf{left},q}$ be the two labels that correspond to the qth input bit $r^q_{j+2,m}$ (similarly $\mathsf{label}_0^{\mathsf{right},q}, \mathsf{label}_1^{\mathsf{right},q}$) are the labels that correspond to $r^q_{j+2,m+1}$). Let $k = r_{j+1,\ell+1-b}$, then, if $r^q_{j+2,m} = \beta$ then the evaluator should learn $\mathsf{label}_\beta^{\mathsf{left},q}$ and input it to the next circuit, therefore we give the evaluator the mapping $\mathsf{translate}^q_{\mathsf{left}} = (\mathsf{ct}^q_{\mathsf{left},0}, \mathsf{ct}^q_{\mathsf{left},1})$ where

$$\mathsf{ct}^q_{\mathsf{left},0} = F_k(\mathsf{left}, 0, q) \oplus \mathsf{label}_0^{\mathsf{left},q} \text{ and } \mathsf{ct}^q_{\mathsf{left},1} = F_k(\mathsf{left}, 1, q) \oplus \mathsf{label}_1^{\mathsf{left},q}$$

Note that the evaluator cannot learn $\mathsf{label}_{1-\beta}^{\mathsf{left},q}$ without knowing $F_k(\mathsf{left}, 1-\beta, q)$ and it does not know $F_k(\mathsf{left}, 1-\beta, q)$ because it does not have k.

Outputs:

– $\widehat{r_{j+1,\ell}}$ and $\widehat{r_{j+1,\ell+1}}$. The evaluator writes these ciphertexts to the ℓ and $\ell+1$ locations, respectively, in the $(j+1)$th layer of the garbled memory.

– The values L and state are passed (in wire labels) directly to the next circuit.

– The key $\tilde{r}$ is passed (via wire labels) to the next circuit and corresponds as the input s (see the above description).

– The 2κ translation tables: $\mathsf{translate}^q_{\mathsf{left}}$ and $\mathsf{translate}^q_{\mathsf{right}}$ for all $q \in [\kappa]$.

The step Circuit

Recall that a 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 Lth 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.

Inputs:

– **Two (plain) data items** D_L, D_{L+1} (where $L \in [n]$) from level d of the tree (this amounts to 2κ input wires). We stress that the circuit can see these keys but the evaluator sees only wire labels that hide them.

- **The key** $s = r_{d-1,L/2}$. That is, the parent key of D_L, D_{L+1}. This amounts to κ input wires.
- **The root's key** ρ used in order to compute a translation table so the evaluator may input $r_{1,0}$ and $r_{1,1}$ to the next navigation circuit. This input amounts to κ input wires.
- **The current state** of the program (which travels until the next cpu-step circuit).
- **The index** L of the destination memory entry to access.

Hardwired values:

- **A set of** 4κ **input wire labels** of the next circuit (the next circuit is $\mathsf{nav}_{i+1,0}$). Those wire labels correspond to two keys $r_{1,0}, r_{1,1}$ in the 1-st level of the memory. Again, there are 4κ labels because there are 2 labels (for zero and one) for each bit of those values. As before, denote those labels by $\mathsf{label}_0^{\mathsf{left},q}, \mathsf{label}_1^{\mathsf{left},q}, \mathsf{label}_0^{\mathsf{right},q}, \mathsf{label}_1^{\mathsf{right},q}$ for all $q \in [\kappa]$.

Procedure:

- Let $L_d = b$ ($b = 0$ indicates that we should use D_L and $b = 1$ indicates that we should use D_{L+1}).
- Compute $(\mathsf{state}', L', D') = C_{\mathsf{CPU}}^P(\mathsf{state}, D_{L+b})$ and re-assign $D_{L+b} \leftarrow D'$.
- Compute $\tilde{D}_0, \tilde{D}_1$ where

$$\tilde{D}_b = F_s(\mathsf{side}, D_{L+b}^1, \kappa), \ldots, F_s(\mathsf{side}, D_{L+b}^{\kappa}, \kappa)$$

  where $\mathsf{side} = \mathsf{left}$ if $b = 0$ and $\mathsf{side} = \mathsf{right}$ if $b = 1$.
- Let $r_{1,0}$ and $r_{1,1}$ be the child keys of the root key ρ. Recall that the evaluator has the encryptions $F_\rho(\mathsf{left}, r_{1,0}^q, q)$ and $F_\rho(\mathsf{right}, r_{1,1}^q, q)$ for all $q \in [\kappa]$ in its garbled memory, and in the next navigation circuit should receive $r_{1,0}^q$ and $r_{1,1}^q$ as inputs. Also recall that the input wire labels by which these 2κ bits are transferred to $\mathsf{nav}_{i+1,0}$ are hardwired and known to the current circuit. Thus, let $\mathsf{label}_0^{\mathsf{left},q}, \mathsf{label}_1^{\mathsf{left},q}$ be the two labels that correspond to the qth input bit $r_{1,0}^q$ (similarly $\mathsf{label}_0^{\mathsf{right},q}, \mathsf{label}_1^{\mathsf{right},q}$) are the labels that correspond to $r_{1,1}^q$). If $r_{1,0}^q = b$ then the evaluator should learn $\mathsf{label}_0^{\mathsf{left},q}$ and input it to the next circuit, therefore we give the evaluator $\mathsf{translate}_{\mathsf{left}}^q = (\mathsf{ct}_{\mathsf{left},0}^q, \mathsf{ct}_{\mathsf{left},1}^q)$ where

$$\mathsf{ct}_{\mathsf{left},0}^q = F_\rho(\mathsf{left}, 0, q) \oplus \mathsf{label}_0^{\mathsf{left},q} \quad and \quad \mathsf{ct}_{\mathsf{left},1}^q = F_\rho(\mathsf{left}, 1, q) \oplus \mathsf{label}_1^{\mathsf{left},q}$$

Outputs:

- $\tilde{D}_0$ and $\tilde{D}_1$. The evaluator writes those encryptions to the L and $L+1$ locations, respectively, in the dth layer of the garbled memory.
- The values L' and state' are passed (in wire labels) directly to the next circuit.
- the 2κ translation tables: $\mathsf{translate}_{\mathsf{left}}^q$ and $\mathsf{translate}_{\mathsf{right}}^q$ for all $q \in [\kappa]$.

A Chain of Circuits

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 key that was refreshed in the prior circuit. An evaluation example of a program with two CPU-steps and 8 data items is presented in Appendix C.

5.2 2PC in the Presence of Malicious Adversaries Relying on OWF

On a high-level, we compile the GRAM from [15] into a malicious two-party protocol using the cut-and-choose approach. Similarly to our protocol from Section 4.1, we extract the randomness that is used for the read and write operations. We note that following this path requires carefully understanding the details of the underlying GRAM scheme which is not a straightforward extension of our protocol from Section 4. Specifically, the security proof for the case of corrupted receiver does not use the GRAM in a black-box way as it depends on the mechanism that prevents rollback (e.g., TIBE or a PRF tree). More specifically, 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 all copies of the evaluation circuits must 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. Specifically, the new PRF keys allow generating the new translation tables to be written in the garbled memory. We continue with the following high level description of the malicious secure protocol.

5.2.1 Protocol $\tilde{\Pi}^{P}_{\text{UMA}}$

Intuitively, the protocol that relies on the existence of one way functions is the same as the protocol described in Section 4.1 with the modifications 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 overview these changes in the following description.

- **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 correspond to the fresh PRF 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 $T \cdot d$ 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 evaluating CPU-step t is a step circuit step_t whereas the rest $d-1$ garbled circuits are navigation circuits $\mathsf{nav}_{t,0}, \ldots, \mathsf{nav}_{t,d-2}$. The functionalities of these circuits were described above. For each time step $t = T, \ldots, 1$, the garbling procedure starts by garbling step_t, then garbling $\mathsf{nav}_{t,d-2}$ and so on till the garbling of $\mathsf{nav}_{t,0}$. Such that the labels associated with the input wires of $\mathsf{nav}_{t,j}$, that represent the current state, are hardwired into circuit $\mathsf{nav}_{t,j-1}$ and similarly, the labels of the input wires of step_t, that represent the current state, are hardwired into circuit $\mathsf{nav}_{t,d-2}$. This is done in the same manner as in the protocol described in Section 4.1.

- **The bootstrapping circuit.** The bootstrapping circuit (C_{INIT}) inputs the values $s = u_0^1$ and $\tilde{r} = u_1^1$ to the first navigation circuit $\mathsf{nav}_{1,0}$, the values $s = u_1^1$ and $\tilde{r} = u_2^1$ to the second navigation circuit

$\mathsf{nav}_{1,1}$ and so on, until reaching the first step circuit step_1 which is given the keys u^1_{d-2}, u^0_0. Generally speaking, the bootstrapping circuit transfers the navigation circuit $\mathsf{nav}_{i,j}$ the inputs $s = u^i_j$ and $\tilde{r} = u^i_{j+1}$, and the circuit step_i the inputs $s = u^i_{d-2}$ and $\tilde{r} = u^i_0$. Note that each key is input to exactly two circuits and after being used to "decrypt" some labels we update it and it is never used again.

We proceed with a formal description of the protocol:

Formal description of protocol $\tilde{\Pi}^P_{\mathrm{UMA}}$

1. Garbled navigation, step and initialization circuits.
   For $t = T-1, \ldots, 1$

   (a) Garble the CPU-step step_t circuit:

   - If $t = T$ then choose random labels for the labels corresponding to state_{T+1}, otherwise, choose random labels for wires associated with outputs $\tilde{D}_0$, $\tilde{D}_1$, $\mathsf{translate}^q_{\mathsf{left}}$ and $\mathsf{translate}^q_{\mathsf{right}}$ for all $q \in [\kappa]$, in addition, use label for output wires of L' and state' as the labels of the input wires of navigation circuit $\mathsf{nav}_{t+1,0}$.
   - Denote all chosen labels by $\{\mathsf{lbl}^{v,i,t}_{\mathrm{out},b}\}_{v,i,b}$ and continue as follows:
   - Garble step_t by calling

   $$\left(\{GC^i_{\mathsf{step},t}\}_i, \{\mathsf{lbl}^{u,i,t}_{\mathrm{in},b}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, \mathsf{step}_t, \{\mathsf{lbl}^{v,i,t}_{\mathrm{out},b}\}_{v,i,b}; r\right)$$

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

   - Interpret the result labels $\{\mathsf{lbl}^{u,i,t}_{\mathrm{in},b}\}_{u,i,b}$ as the following groups of values: $D_{L,t}$, $D_{L+1,t}$, s_t, ρ, state_t and L, that cover the labels:
   $\{\mathsf{lbl}_{D_{L,t}}\}$, $\{\mathsf{lbl}_{D_{L+1,t}}\}$, $\{\mathsf{lbl}_{s_t}\}$, $\{\mathsf{lbl}_{\rho}\}$, $\{\mathsf{lbl}_{\mathsf{state}_t}\}$ and $\{\mathsf{lbl}_L\}$, respectively.

   (b) Garble the navigation circuits:
   For $j = d-2, \ldots, 0$:

   - If $j = d-2$ then hardwire the labels $\{\mathsf{lbl}_{D_{L,t}}\}$ and $\{\mathsf{lbl}_{D_{L+1,t}}\}$ within $\mathsf{nav}_{t,j}$, otherwise, hardwire the labels $\{\mathsf{lbl}_{\mathsf{key}_{\mathsf{left},t,j+1}}\}$ and $\{\mathsf{lbl}_{\mathsf{key}_{\mathsf{right},t,j+1}}\}$ within $\mathsf{nav}_{t,j}$
   - Choose random labels for the output wires that correspond to $\widehat{r_{j+1,\ell}}$ and $\widehat{r_{j+1,\ell+1}}$ and $\mathsf{translate}^q_{\mathsf{left}}$ and $\mathsf{translate}^q_{\mathsf{right}}$ for all $q \in [\kappa]$, in addition, use labels for output wires of L and state as the labels of the input wires of navigation circuit $\mathsf{nav}_{t,j+1}$.
   - Denote all chosen labels by $\{\mathsf{lbl}^{v,i,t}_{\mathrm{out},b}\}_{v,i,b}$ and continue as follows:
   - Garble $\mathsf{nav}_{t,j}$ by calling

   $$\left(\{GC^i_{\mathsf{nav},t,j}\}_i, \{\mathsf{lbl}^{u,i,t}_{\mathrm{in},b}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, \mathsf{nav}_{t,j}, \{\mathsf{lbl}^{v,i,t}_{\mathrm{out},b}\}_{v,i,b}; r\right)$$

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

   - Interpret the result labels $\{\mathsf{lbl}^{u,i,t}_{\mathrm{in},b}\}_{u,i,b}$ as the following groups of values: $\{\mathsf{lbl}_{\mathsf{key}_{\mathsf{left},t,j}}\}$, $\{\mathsf{lbl}_{\mathsf{key}_{\mathsf{right},t,j}}\}$, $\{\mathsf{lbl}_{s_{t,j}}\}$, $\{\mathsf{lbl}_{\tilde{r}_{t,j}}\}$, $\{\mathsf{lbl}_{\mathsf{state}_t}\}$ and $\{\mathsf{lbl}_L\}$, respectively.

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

- Combine the group of labels $\{\mathsf{lbl}_{s_{t,j}}\}$ and $\{\mathsf{lbl}_{\tilde{r}_{t,j}}\}$ in addition to the input wires labels from the first navigation circuit $\mathsf{nav}_{0,0}$ that correspond to the state and denote them by $\{\mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b}$.
- Garble the initialization circuit:

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

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

2. OBLIVIOUS TRANSFERS.
This step goes exactly as in protocol Π^{P}_{UMA}.

3. SEND GARBLED CIRCUITS.
S sends R the garbled circuits chains $\mathrm{GC}^{i}_{\mathrm{INIT}}, \mathsf{step}^{i}_{t}, \mathsf{nav}^{i}_{t,j}$ for every $t \in [T], j \in [d-1]$ and $i \in [s]$.

4. COMMITMENTS AND CUT-AND-CHOOSE CHALLENGE.
This step goes exactly as in protocol Π^{P}_{UMA}.

5. SEND ALL INPUT GARBLED VALUES IN CHECK CIRCUITS.
This step goes exactly as in protocol Π^{P}_{UMA}.

6. CORRECTNESS OF CHECK CIRCUITS.
This step goes exactly as in protocol Π^{P}_{UMA}.

7. CHECK GARBLED INPUTS CONSISTENCY FOR THE EVALUATION-CIRCUITS.
This step goes exactly as in protocol Π^{P}_{UMA}.

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 $\mathrm{GC}^{z}_{\mathrm{INIT}}$ using Eval and the input wires it obtained in Step 7 and reveal one label for each of its output wires $\mathsf{lbl}^{\mathrm{out},z}_{\mathrm{INIT}}$.
   These output wires correspond to two keys for every of the next circuits as described above.

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

      i. For $j = 0, \ldots, d-2$ evaluate the circuit $GC^{z}_{\mathsf{nav},t,j}$ for all $z \in \tilde{Z}$. As in Π^{P}_{UMA}, take the majority across the results of all circuits and write it to the appropriate location in the garbled memory tree (i.e. the values written to the gabled memory).

      ii. Evaluate $GC^{z}_{\mathsf{step},t}$ for all $z \in \tilde{Z}$ and again take the majority and use it to write to the appropriate location in the garbled memory tree. If $t = T$, output state_{T} in the clear.

We prove the following theorem:

Theorem 5.1. *Assume the existence of one-way functions, π_{GC} is a garbling scheme (cf. Definition 2.2), and* Com *is a statistical binding commitment scheme (cf. Definition A.1). Then, protocol $\tilde{\Pi}^{P}_{\mathrm{UMA}}$ (cf. Section 5.2.1) securely realizes $\mathcal{F}_{\mathrm{UMA}}$ in the presence of malicious adversaries in the $\{\mathcal{F}_{\mathrm{SCCOT}}, \mathcal{F}_{\mathrm{IC}}\}$-hybrid models. In addition, all asymptotic complexities are as implied by Theorem 4.1.*

Proof Sketch. Note first that the above nav and step circuits are given two random PRF keys as inputs from the bootstrapping circuit $\mathrm{C}_{\mathrm{INIT}}$ where these two PRF keys are used in the same way as in the original protocol description. Intuitively, the following two arguments hold: (a) Correctness. Namely, 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) Privacy. The evaluator does not learn anything beyond the program output and the memory access pattern. Where the security analysis and the constructions of the simulators follow similarly to the proof of Theorem 4.1.

The case S **is corrupted.** Namely, in case the sender is corrupted, then the cut-and-choose analysis ensures that the majority of the garbled circuits have been correctly constructed. Where the view of the sender can be simulated similarly to the previous section as it is, where before the functionality within Π^{P}_{UMA} is embedded with the descriptions of the TIBE algorithms whereas $\tilde{\Pi}^{P}_{\mathrm{UMA}}$ implements PRF evaluations.

The case R **is corrupted.** On the other hand, in case the receiver is corrupted, the sender's privacy is ensured by the underlying GRAM construction which prevents from the receiver to rollback or run multiple executions on several memory instances. In more details, the simulator produces simulated garbled circuits starting from the last circuit. It proceeds by generating a random-looking output for each of the circuits by setting the $\mathsf{translate}$ tables to be random keys XORed with the corresponding input labels of the next step or nav circuit (since we are working backwards, these labels have already been generated), and similarly using random values for emulating the write operations.

The main idea is that we keep track of these random values so that when we simulate the garbled database, we set $\tilde{D}$ to be uniformly random subject to the stored elements. Now, since the simulator gets the full access pattern, it knows exactly which locations in memory it should set entries for, so that they match values that were used to mask the translation table. This scheme was proven secure in [15] for the case of a semi-honest adversary whereas we claim security against a malicious adversary. However, we stress that since we are working in the $\mathcal{F}_{\mathrm{IC}}$-hybrid model, which ensures input consistency, the adversary's inputs are extracted and the rest of the simulation goes exactly as theirs which concludes the proof.

References

[1] A. Afshar, Z. Hu, P. Mohassel, and M. Rosulek. How to efficiently evaluate RAM programs with malicious security. In *EUROCRYPT*, pages 702–729, 2015.

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

[3] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, pages 503–513, 1990.

[4] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *CCS*, pages 784–796, 2012.

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

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

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

[8] T. H. Chan and E. Shi. Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In *TCC*, pages 72–107, 2017.

[9] K. Chung and R. Pass. A simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.

[10] S. A. Cook and R. 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.

[11] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.

[12] J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *CCS*, pages 523–535, 2017.

[13] S. Garg, D. Gupta, P. Miao, and O. Pandey. Secure multiparty RAM computation in constant rounds. In *TCC*, pages 491–520, 2016.

[14] S. Garg, S. Lu, and R. Ostrovsky. Black-box garbled RAM. In *FOCS*, pages 210–229, 2015.

[15] S. Garg, S. Lu, R. Ostrovsky, and A. Scafuro. Garbled RAM from one-way functions. In *STOC*, pages 449–458, 2015.

[16] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PETS*, pages 1–18, 2013.

[17] C. Gentry, S. Halevi, C. S. Jutla, and M. Raykova. Private database access with he-over-oram architecture. In *ACNS*, pages 172–191, 2015.

[18] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled RAM revisited. In *EUROCRYPT*, pages 405–422, 2014.

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

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

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

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

[23] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, pages 157–167, 2012.

[24] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, pages 513–524, 2012.

[25] Z. Hu, P. Mohassel, and M. Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In *CRYPTO*, pages 150–169, 2015.

[26] Y. Ishai, E. Kushilevitz, R. Ostrovsky, M. Prabhakaran, and A. Sahai. Efficient non-interactive secure computation. In *EUROCRYPT*, pages 406–425, 2011.

[27] Y. Ishai, M. Prabhakaran, and A. Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, pages 572–591, 2008.

[28] Y. Ishai, M. Prabhakaran, and A. Sahai. Secure arithmetic computation with no honest majority. In *TCC*, pages 294–314, 2009.

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

[30] M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. In *ASIACRYPT*, pages 506–525, 2014.

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

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

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

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

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

[36] C. Liu, Y. Huang, E. Shi, J. Katz, and M. W. Hicks. Automating efficient ram-model secure computation. In *IEEE Symposium on Security and Privacy*, pages 623–638, 2014.

[37] S. Lu and R. Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, pages 719–734, 2013.

[38] P. Miao. Cut-and-choose for garbled RAM. *IACR Cryptology ePrint Archive*, 2016:907, 2016.

[39] S. Micali and P. Rogaway. Secure computation (abstract). In *CRYPTO*, pages 392–404, 1991.

[40] J. B. Nielsen and C. Orlandi. Lego for two-party secure computation. In *TCC*, pages 368–386, 2009.

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

[42] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, pages 250–267, 2009.

[43] N. Pippenger and M. J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[44] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX*, pages 415–430, 2015.

[45] E. Shi, T. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with o((logn)3) worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[46] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, pages 299–310, 2013.

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

[48] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi. SCORAM: oblivious RAM for secure computation. In *CCS*, pages 191–202, 2014.

[49] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *CCS*, pages 293–304, 2012.

[50] A. C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.

[51] A. C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

[52] S. Zahur, X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *SP*, pages 218–234, 2016.

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 2.1. In this section we present the Input Consistency Functionality (Figure 6) 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 7) used in the definition of garbled circuits.

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

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 $|x| = v_{\text{in}}$.

Common inputs.

- The circuit C and the security parameters κ, s.
- s garbled versions of C, namely $\{\tilde{\mathrm{C}}_i\}_{i\in[s]}$.
- s sets of garbled input $\left\{\left(\mathsf{lbl}^{1,i}_{\text{in},x[1]},\ldots,\mathsf{lbl}^{v_{\text{in}},i}_{\text{in},x[v_{\text{in}}]}\right)\right\}_{i\in[s]}$.
- s sets of commitments for the sender's input labels, denoted by $\{\mathsf{com}^i_{1,b},\ldots,\mathsf{com}^i_{v_{\text{in}},b}\}_{b\in\{0,1\},i\in[s]}$.

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

- The output labels used in Garb, denoted by $\{\mathsf{lbl}^{v,i}_{\text{out},b}\}_{v,i,b}$.
- The randomness r used in Garb.
- Decommitments $\{\mathsf{dec}^i_{1,b},\ldots,\mathsf{dec}^i_{v_{\text{in}},b}\}_{b\in\{0,1\},i\in[s]}$ for the above commitments to the input labels.

Output. The functionality works as follows:

- Compute
$$\left(\{\hat{\mathrm{C}}_i\}_i,\{u,b,\hat{\mathsf{lbl}}^{u,i}_{\text{in},b}\}_{u,i,b}\right)\leftarrow\mathsf{Garb}\left(1^\kappa,s,\mathrm{C},\{v,b,\mathsf{lbl}^{v,i}_{\text{out},b}\}_{v,i,b};r\right)$$
- For every $u\in[v_{\text{in}}]$:
  - For every $i\in[s]$ set b_i as
$$b_i=\left\{\begin{array}{ll}0, & \mathsf{com}(\mathsf{lbl}^{u,i}_{\text{in},x[u]},\mathsf{dec}^i_{u,0})=\mathsf{com}^i_{u,0}\\ 1, & \mathsf{com}(\mathsf{lbl}^{u,i}_{\text{in},x[u]},\mathsf{dec}^i_{u,1})=\mathsf{com}^i_{1,0}\\ \perp & \text{otherwise}\end{array}\right\}$$
  - If $b_i=\perp$ 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.

Figure 6: The input consistency functionality $\mathcal{F}_{\text{IC}}$.

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 $\{v, b, \mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b}$ for every $v \in [v_{\mathrm{out}}], i \in [s]$ and $b \in \{0,1\}$ and computes:
$$\left(\{\tilde{\mathrm{C}}_i\}_i, \{u, b, \mathsf{lbl}^{u,i}_{\mathrm{in},b}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\Big(1^\kappa, s, \mathrm{C}, \{v, b, \mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b}\Big)$$
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 $\tilde{x}_z = (\mathsf{lbl}^{1,z}_{\mathrm{in},b}, \ldots, \mathsf{lbl}^{v_{\mathrm{in}},z}_{\mathrm{in},b})$ for every $b \in \{0,1\}$; while for garbled circuits indexed with $z \notin Z$ the set is $\tilde{x}_z = (\mathsf{lbl}^{1,z}_{\mathrm{in},x[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{in}},z}_{\mathrm{in},x[v_{\mathrm{in}}]})$ for some input x.
4. The adversary returns a single index z and one set of output labels $\hat{y}_z = (\hat{\mathsf{lbl}}^{1,z}_{\mathrm{out},b}, \ldots, \hat{\mathsf{lbl}}^{v_{\mathrm{out}},z}_{\mathrm{in},b})$.
5. The game concludes as follows:
   (a) If $z \in Z$ return 0. Otherwise continue.
   (b) Compute $(\mathsf{lbl}^{1,z}_{\mathrm{out},y[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{out}},z}_{\mathrm{in},y[v_{\mathrm{out}}]}) = \mathsf{Eval}(\tilde{\mathrm{C}}_z, \tilde{x}_z)$.
   (c) If for some $j \in [v_{\mathrm{out}}]$ it holds that $\hat{\mathsf{lbl}}^{j,z}_{\mathrm{out},b} = \mathsf{lbl}^{1,z}_{\mathrm{out},1-y[1]}$ then output 1. Otherwise, output 0.

Figure 7: The authenticity game $\mathrm{Auth}_{\mathcal{A}}(1^\kappa, s, \mathrm{C})$.

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 computing $\mathcal{F}$. By the composition theorem [7] 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 8

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

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 8: Batch single choice cut-and-choose OT $\mathcal{F}_{\text{SCCOT}}$.

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.*

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

where $\mathbf{View}^{\mathrm{R}^*}_{\mathsf{Com}}(m, z)$ *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 Definition 2.2

In this section we argue that our definition of garbled circuit with respect to cut-and-choose based protocols (cf. Definition 2.2) can be realized by the garbling scheme described in [35]. 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 9 the garbling procedure for a *single* circuit; then, in Figure 10 we describe the full garbling procedure Garb that uses GarbYao as a sub-procedure.

Procedure GarbYao

Input.

- A circuit description $\mathrm{C}_{\mathrm{CPU}}$ with v_{in} (resp. v_{out}) input (resp. output) wires, and a total of W wires.
- A set of v_{out} output labels $\mathsf{lbl}^0_{\mathrm{out},1}, \mathsf{lbl}^1_{\mathrm{out},1}, \ldots, \mathsf{lbl}^0_{\mathrm{out},v_{\mathrm{out}}}, \mathsf{lbl}^1_{\mathrm{out},v_{\mathrm{out}}}$.
- A set of input labels $\mathsf{lbl}^0_{\mathrm{in},1}, \mathsf{lbl}^1_{\mathrm{in},1}, \ldots, \mathsf{lbl}^0_{\mathrm{in},v_{\mathrm{in}}}, \mathsf{lbl}^1_{\mathrm{in},v_{\mathrm{in}}}$.

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. [33]). For each gate $G \in \mathrm{C}_{\mathrm{CPU}}$ with input wires a, b and output wire c such that G computes the binary function $g : \{0,1\}^2 \rightarrow \{0,1\}$, 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 $\widetilde{\mathrm{C}_{\mathrm{CPU}}} = \bigcup_{G \in \mathrm{C}_{\mathrm{CPU}}} \tilde{G}$.

Figure 9: Procedure GarbYao for a single circuit garbling.

Evaluation. As modulated in the garbling procedure, we first show how a single garbled circuit can be evaluated in Figure 11 and then in Figure 12 we show how, using EvalYao as a sub-procedure, we evaluate a set of s garbled circuits.

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 $\mathrm{C}_{\mathrm{CPU}}$ with v_{in} (resp. v_{out}) input (resp. output) wires, and a total of W wires.
- s set of v_{out} output labels $\mathsf{lbl}^{0,i}_{\mathrm{out},1}, \mathsf{lbl}^{1,i}_{\mathrm{out},1}, \ldots, \mathsf{lbl}^{0,i}_{\mathrm{out},v_{\mathrm{out}}}, \mathsf{lbl}^{1,i}_{\mathrm{out},v_{\mathrm{out}}}$ for all $i \in [s]$.

Output.

- Let $[\ell] \subset [v_{\mathrm{in}}]$ 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:

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

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

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

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

Figure 10: Procedure Garb for s circuits garbling.

Correctness and privacy. The correctness and privacy properties had been proven in [33] for a single circuit. It is trivial to show that these properties hold in the cut-and-choose notion we defined.

Authenticity. The authenticity property is missing from [18] 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 [33] 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 $\mathrm{C}_{\mathrm{CPU}}$ and an adversary $\mathcal{A}$, for which $\Pr[\mathrm{Auth}_{\mathcal{A}}(1^\kappa, s, \mathrm{C}) = 1] = p$, 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 $\widetilde{\mathrm{C}_{\mathrm{CPU}}}$ and a garbled input $\tilde{x}$ for a given input x such that $\widetilde{\mathrm{C}_{\mathrm{CPU}}}(\tilde{x}) = \tilde{y}$. $\mathcal{D}$ hands $\widetilde{\mathrm{C}_{\mathrm{CPU}}}, x, \tilde{x}$ 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.

Procedure $\mathsf{EvalYao}$

Inputs.

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

Output.

- For every garbled gate in the set of garbled gates in $\widetilde{\mathrm{C}_{\mathrm{CPU}}}$ (in a topological order). Let a, b be its input wires, $\mathsf{lbl}^\alpha_a, \mathsf{lbl}^\beta_b$ 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 $\mathsf{pt}_t = \mathsf{Dec}_{\mathsf{lbl}^\beta_b}(\mathsf{Dec}_{\mathsf{lbl}^\alpha_a}(\mathsf{ct}_t))$.
  - Set $\mathsf{lbl}^\gamma_c = \mathsf{pt}_t$ for the only t for which $\mathsf{pt}_t \neq \perp$. (There is always a single pt_t except with negligible probability).
- Output $(\mathsf{lbl}^1_{\mathrm{out},y[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{out}}}_{\mathrm{out},y[v_{\mathrm{out}}]})$ for some output y.

Figure 11: Procedure $\mathsf{EvalYao}$ for a single garbled circuit.

Procedure Eval

Inputs.

- s garbled circuits $\{\widetilde{\mathrm{C}_{\mathrm{CPU}i}}\}_{i\in[s]}$.
- s sets of v_{in} labels $\big\{(\mathsf{lbl}^{1,i}_{\mathrm{in},x[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{in}},i}_{\mathrm{in},x[v_{\mathrm{in}}]})\big\}_{i\in[s]}$ for some input x.

Output.

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

$$(\mathsf{lbl}^{1,i}_{\mathrm{out},x[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{out}},i}_{\mathrm{out},y[v_{\mathrm{out}}]}) \leftarrow \mathsf{EvalYao}\big(\widetilde{\mathrm{C}_{\mathrm{CPU}}}, (\mathsf{lbl}^{1,i}_{\mathrm{in},x[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{in}},i}_{\mathrm{in},x[v_{\mathrm{in}}]})\big)$$

- Output $\big\{(\mathsf{lbl}^{1,i}_{\mathrm{out},y[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{out}},i}_{\mathrm{out},y[v_{\mathrm{out}}]})\big\}_{i\in[s]}$ for some output y.

Figure 12: Procedure Eval to evaluate s garbled circuits.

Input consistency. We now show the protocol that realizes the input consistency functionality $\mathcal{F}_{\mathrm{IC}}$ from Figure 6 with respect to the garbling scheme $(\mathsf{Garb}, \mathsf{Eval})$ from above. The common inputs are

- Security parameters s, κ.
- The circuit $\mathrm{C}_{\mathrm{CPU}}$ and s garbled versions $\{\widetilde{\mathrm{C}_{\mathrm{CPU}i}}\}_{i\in[s]}$. Note that s here is a subset of the s which the sender used in the garbling phase.
- Labels $(\mathsf{lbl}^i_1, \ldots, \mathsf{lbl}^i_\ell) = (g^{a_1^{x[1]}\cdot r_i}, \ldots, g^{a_\ell^{x[\ell]}\cdot r_i})$ 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 $\{g^{a_j^{x[j]} \cdot r_i}\}_{i \in [s]}$ is consistent: For every $j \in [\ell]$ the sender uses the protocol in Figure 13 to prove that there exist a value $\sigma_j \in \{0,1\}$ such that for every $i \in [s]$, $\mathsf{lbl}_j^i = g^{a_1^{\sigma_j} \cdot r_i}$. 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 $\perp$.

For completeness, we provide the protocol, used in [35], verbatim.

ZK proof for extended Diffie-Hellman tuples. A zero-knowledge proof of an extended Diffie-Hellman tuple is given in Figure 13. The input is a tuple $(g, h_0, h_1, u_1, v_1, \ldots, u_\eta, v_\eta)$ such that either all $\{(g, h_0, u_i, v_i)\}_{i=1}^{\eta}$ are Diffie-Hellman tuples, or all $\{(g, h_1, u_i, v_i)\}_{i=1}^{\eta}$ are Diffie-Hellman tuples. It is shown in [35] that the protocol in Figure 13 is a ZK-PoK.

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 [35] for more details).

Figure 13: ZK Proof of knowledge of extended Diffie-Hellman tuples.

C Program Execution Example

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):

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

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

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

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

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:

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

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 $\mathsf{nav}_{2,0}$.

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

$$\tilde{D} = \Big\{ F_{u_0}(v_1, \mathsf{left}), F_{u_0}(u_1, \mathsf{right}), F_{v_1}(r_{2,0}, \mathsf{left}), F_{v_1}(v_2, \mathsf{right}), F_{r_{1,1}}(r_{2,2}, \mathsf{left}), F_{r_{1,1}}(r_{2,3}, \mathsf{right}), F_{r_{2,0}}(D_0, \mathsf{left}), F_{r_{2,0}}(D_1, \mathsf{right}), F_{v_2}(D_2, \mathsf{left}), F_{v_2}(D_3, \mathsf{right}), F_{r_{2,2}}(D_4, \mathsf{left}), F_{r_{2,2}}(D_5, \mathsf{right}), F_{r_{2,3}}(D_6, \mathsf{left}), F_{r_{2,3}}(D_7, \mathsf{right}) \Big\}$$

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

$$\tilde{D} = \Big\{ F_{u_0}(v_1, \mathsf{left}), F_{u_0}(u_1, \mathsf{right}), F_{v_1}(r_{2,0}, \mathsf{left}), F_{v_1}(v_2, \mathsf{right}), F_{u_1}(r_{2,2}, \mathsf{left}), F_{u_1}(u_2, \mathsf{right}), F_{r_{2,0}}(D_0, \mathsf{left}), F_{r_{2,0}}(D_1, \mathsf{right}), F_{v_2}(D_2, \mathsf{left}), F_{v_2}(D_3, \mathsf{right}), F_{r_{2,2}}(D_4, \mathsf{left}), F_{r_{2,2}}(D_5, \mathsf{right}), F_{r_{2,3}}(D_6, \mathsf{left}), F_{r_{2,3}}(D_7, \mathsf{right}) \Big\}$$

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

$$\tilde{D} = \Big\{ F_{u_0}(v_1, \mathsf{left}), F_{u_0}(u_1, \mathsf{right}), F_{v_1}(r_{2,0}, \mathsf{left}), F_{v_1}(v_2, \mathsf{right}), F_{u_1}(r_{2,2}, \mathsf{left}), F_{u_1}(u_2, \mathsf{right}), F_{r_{2,0}}(D_0, \mathsf{left}), F_{r_{2,0}}(D_1, \mathsf{right}), F_{v_2}(D_2, \mathsf{left}), F_{v_2}(D_3, \mathsf{right}), F_{r_{2,2}}(D_4, \mathsf{left}), F_{r_{2,2}}(D_5, \mathsf{right}), F_{u_2}(D_6, \mathsf{left}), F_{u_2}(D_7, \mathsf{right}) \Big\}$$

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