Constant-Round Maliciously Secure Two-Party Computation in the RAM Model ⋆ ⋆⋆

Carmit Hazay and Avishay Yanai

Bar Ilan University, Israel
carmit.hazay@biu.ac.il, ay.yanay@gmail.com

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. [16] 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. [13] that is based on one-way functions.

1 Introduction

Background on secure computation. Secure multi-party computation enables a set of parties to mutually run a protocol that computes some function f on their private inputs, while preserving a number of security properties. Two of the most important properties are privacy and correctness. The former implies data confidentiality, namely, nothing leaks by the protocol execution but the computed output. The latter requirement implies that the protocol enforces the integrity of the computations made by the parties, namely, honest parties learn the correct output. More generally, a rigorous security definition requires that distrusting parties with secret inputs will be able to compute a function of their inputs as if the computation is executed in an ideal setting, where the parties send their inputs to a incorruptible trusted party that performs the computation and returns its result (also known by the ideal/real paradigm). The feasibility of secure computation has been established by a sequence of works [48, 19, 2, 36, 7], proving security under

⋆ 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).

⋆⋆ This paper was presented jointly with [11] in proceedings of the 14th IACR Theory of Cryptography Conference (TCC) 2016-B.

this rigorous definition with respect to two adversarial models: the semi-honest model (where the adversary follows the instructions of the protocol but tries to learn more than it should from the protocol transcript), and the malicious model (where the adversary follows an arbitrary polynomial-time strategy).

Following these works much effort was put in order to improve the efficiency of computation with the aim of minimizing the workload of the parties [27, 31, 25, 26, 39, 37, 32, 24, 30]. These general-purpose protocols are restricted to functions represented by Boolean/arithmetic circuits. Namely, the function is first translated into a (typically Boolean) circuit and then the protocol securely evaluates it gate-by-gate on the parties' private inputs. This approach, however, falls short when the computation involves access to a large memory since in the circuits-based approach, dynamic memory accesses, which depend on the secret inputs, are translated into a linear scan of the memory. This translation is required for every memory access and causes a huge blowup in the description of the circuit.

The RAM model of computation. We further note that the majority of applications encountered in practice today are more efficiently captured using *random-access memory (RAM)* programs that allow constant-time memory lookup. This covers graph algorithms, such as the known Dijkstra's shortest path algorithm, binary search on sorted data, finding the kth-ranked element, the Gale-Shapely stable matching algorithm and many more. This is in contrast to the sequential memory access that is supported by the architecture of Turing machines. Generic transformations from RAM programs that run in time T generate circuits of size $(T^3 \log T)$ which are non-scalable even for cases where the memory size is relatively small [9, 40].

To address these limitations, researchers have recently started to design secure protocols directly in the RAM model [10, 22, 1]. The main underlying idea is to rely on Oblivious RAM (ORAM) [17, 38, 20], 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 as well as the access pattern to it.

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

Secure computation for RAM programs. An important application of ORAM is in gaining more efficient protocols for secure computation [22, 14, 34, 15] [16, 28, 33, 45, 44, 1,

13, 23, 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 whom their round complexity depends on the ORAM running time. In [22] Gordon et al. designed the first rigorous semi-honest secure protocols based on this approach, that achieves sublinear amortized overhead that is asymptotically close to the running time of the underlying RAM program in an insecure environment.

As observed later by Afshar et al. [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 their work is the *feasibility of constant round malicious secure computation in the RAM model*. In this work we address this question in the two-party setting.

1.1 Our Results

We design the first constant round maliciously secure protocol for arbitrary RAM programs. Our starting point is the garbled RAM construction of Gentry et al. [16], which is the analogue object of garbled circuits [47, 4] with respect to RAM programs. Namely, a user can garble an arbitrary RAM program directly without converting it into a circuit first. A garbled RAM scheme can be used to garble the data, the program and the input in a way that reveals only the evaluation outcome and nothing else. In their work, Gentry et al. proposed two ways to fix a subtle point emerged in an earlier construction by Lu and Ostrovsky [34] that requires a complex "circular" use of Yao garbled circuits and PRFs. For simplicity, we chose to focus on their garbled RAM based on identity based encryption (IBE) schemes. We show how to transform their IBE based protocol into a maliciously secure 2PC protocol at the cost of involving the cut-and-choose

technique. Following that, in the full version we show how to achieve the same result using the garbled RAM construction of Garg et al. [13] assuming only the existence of one-way-functions. We state our main theorem below,

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

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

Challenges faced in the malicious setting and RAM programs.

1. MEMORY MANAGEMENT. Intuitively speaking, garbled RAM readily induces a two-party protocol with semi-honest security by exchanging the garbled input using oblivious transfer (OT). The natural approach for enhancing the security of a garbled RAM scheme into a maliciously 2PC protocol is by using the cut-and-choose approach [31] where the basic underlying semi-honest protocol is repeated s times (for some statistical parameter s), such that a subset of these instances are "opened" in order to demonstrate correct behaviour whereas the rest of "unopened" instances are used to obtaining the final outcome (typically by taking the majority of results). The main challenge in boosting the security of a semi-honest secure protocol into the malicious setting, using this technique in the RAM model, is with handling multiple instances of memory data. That is, since each semi-honest protocol instance is executed independently, the RAM program implemented within this instance is associated with its own instance of memory. Recalling that the size of the memory might be huge compared to the other components in the RAM system, it is undesirable to store multiple copies of the data in the local memory of the parties. Therefore, the first challenge we had to handle is how to work with multiple copies of the same protocol while having access to a single memory data.
2. HANDLING CHECK/EVALUATION CIRCUITS. The second challenge concerns the cut-and-choose proof technique as well. The original approach to garble the memory is by using encryptions computed based on PRF keys that are embedded inside the garbled circuits. These keys are used to generate a translation mapping which allows the receiver to translate between the secret keys and the labels of the read bit in the next circuit. When employing the cut-and-choose technique, all the secret information embedded within the circuits is exposed during the check process of that procedure which might violate the privacy of the sender. The same difficulty arises when hardwiring the randomness used for the encryption algorithm. A naive solution would be to let the sender choose s sets of keys, such that each set is used

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

within the appropriate copy of the circuit. While this solution works, it prevents the evaluator from determining the majority of the (intermediate) results of all copies.

3. INTEGRITY AND CONSISTENCY OF MEMORY OPERATIONS. During the evaluation of program P, the receiver reads and writes back to the memory. In the malicious setting these operations must be backed up with a mechanism that enforces correctness. Moreover, a corrupted evaluator should not be able to roll back the stored memory to an earlier version. This task is very challenging in a scenario where the evaluator locally stores the memory and fully controls its accesses without the sender being able to verify whether the receiver has indeed carried out the required instructions (as that would imply that the round complexity grows linearly with the running time of the RAM program).

Constant round 2PC in the RAM model. Towards achieving malicious security, we demonstrate how to adapt the garbled RAM construction from [16] into the two-party setting while achieving malicious security. Our protocol is combined of two main components. First, an initialization circuit is evaluated in order to create all the IBE keys (or the PRF keys) that are incorporated in the latter RAM computation, based on the joint randomness of the parties (this phase is not computed locally since we cannot rely on the sender properly creating these keys). Next, the program P is computed via a sequence of small CPU-steps that are implemented using a circuit that takes as input the current CPU state and a bit that was read from the last read memory location, and outputs an updated state, the next location to read, a location to write to and a bit to write into that location. In order to cope with the challenges regarding the cut-and-choose approach, we must ensure that none of the secret keys nor randomness are incorporated into the circuits, but instead given as inputs. Moreover, to avoid memory duplication, all the circuits are given the same sequence of random strings. This ensures that the same set of secret keys/ciphertexts are created within all CPU circuits.

We note that our protocol is applicable to any garbled scheme that supports wire labels and can be optimized using all prior optimizations. Moreover, in a variant of our construction the initialization phase can be treated as a preprocessing phase that does not depend on the input. We further note that our abstraction of garbled circuits takes into account authenticity [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.

As final remark, we note that our construction employs the underlying ORAM in a black-box manner as the parties invoke it locally. This is in contrast to alternative approaches that compute the ORAM using a two (or multi)-party secure protocol such as in [22].

Complexity. The overhead of our protocol is dominated by the complexity induced by the garbled RAM construction of [16] times s, where s is the cut-and-choose statistical parameter. The [16] construction guarantees that the size/evaluation time of the garbled program is $|\mathrm{C}^P_{\mathrm{CPU}}| \times T \times \mathsf{poly}(\kappa) \times \mathsf{polylog}(n)$. Therefore the multiplicative overhead of our protocol is $\mathsf{poly}(\kappa) \times \mathsf{polylog}(n) \times s$.

Reusable/persistent data. Reusable/persistent data means that the garbled memory data can be reused across multiple program executions. That is, all memory updates are persist for future program executions and cannot be rolled back by the malicious evaluator. This feature is very important as it allows to execute a sequence of programs without requiring to initialize the data for every execution, implying that the running time is only proportional to the program running time (in a non-secured environment). The [16] garbled RAM allows to garble any sequence of programs (nevertheless, this set must be given to the garbler in advance and cannot be adaptively chosen). We show that our scheme preserves this property in the presence of malicious attacks as well.

Concurrent work. In a concurrent and independent work by Garg, Gupta, Miao and Pandey [11], the authors demonstrate constant-round multi-party computation with the advantage of achieving a construction that is *black-box* in the one-way function. Their work is based on the black-box GRAM construction of [12] and the constant-round MPC construction of [3]. Their semi-honest secure protocol achieves persistent data, whereas their maliciously secure protocol achieves the weaker notion of selectively choosing the inputs in advance, as we do. The core technique of pulling secrets out of the programs and into the inputs is common to both our and their work. Whereas our construction achieves two features which [12] does not. First, we use the ORAM in a black-box way since the parties can locally compute it. Second, only one party locally stores the memory, rather than both parties string shares of the memory. In another paper [35], Miao demonstrates how to achieve persistent data in the two-party setting assuming a random oracle and using techniques from [37] and [4], where the underlying one-way function is used in a black-box manner.

2 Preliminaries

2.1 The RAM Model of Computation

We follow the notation from [16] verbatim. We consider a program P that has random-access to a memory of size n, which is initially empty. In addition, the program gets a "short" input x, which we can alternatively think of as the initial state of the program. We use the notation $P^D(x)$ to denote the execution of such program. The program can read/write to various locations in memory throughout the execution. [16] also considered the case where several different programs are executed sequentially and the memory persists between executions. Our protocol follows this extension as well. Specifically, this process is denoted as $(y_1, \ldots, y_c) = (P_1(x_1), \ldots, P_\ell(x_c))^D$ to indicate that first $P_1^D(x_1)$ is executed, resulting in some memory contents D_1 and output y_1, then $P_2^{D_1}(x_2)$ is executed resulting in some memory contents D_2 and output y_2 etc.

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

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

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

Predictably Time Writes Predictably Time Writes (ptWrites) means that whenever we want to read some location i in memory, it is easy to figure out the time (i.e., CPU step) t' in which that location was last written to, given only the current state of the computation and without reading any other values in memory. In [16] the authors describe how to upgrade a solution for ptWrites to one that allows arbitrary writes. More formally,

Definition 1 (Predictably timed writes [16]). *A program execution $P^D(x_1, x_2)$ has predictably timed writes if there exists a poly-size circuit, denoted* $\mathsf{WriteTime}$*, such that the following holds for every CPU step $t = 1, \ldots, T$. Let the inputs/outputs of the t-th CPU step be* $\mathsf{cpu\text{-}step}(\mathsf{state}_t, b^{\mathsf{read}}_t) = (\mathsf{state}_{t+1}, i^{\mathsf{read}}_t, i^{\mathsf{write}}_t, b^{\mathsf{write}}_t)$*, then $t' =$* $\mathsf{WriteTime}(t, \mathsf{state}_t, i^{\mathsf{read}}_t)$ *is the largest value of $t' < t$ such that the CPU step t' wrote to memory location i^{read}_t; i.e. $i^{\mathsf{write}}_{t'} = i^{\mathsf{read}}_t$.*

As in [16], we also describe a solution for RAM programs that support ptWrites and then show how to extend it to the general case.

2.2 Oblivious RAM (ORAM)

ORAM, initially proposed by Goldreich and Ostrovsky [17, 38, 20], is an approach for making a read/write memory access pattern of a RAM program input-oblivious. More precisely, it allows a client to store private data on an untrusted server and maintain obliviousness while accessing that data, by only storing a short local state. A secure ORAM scheme not only hides the content of the memory from the server, but also the access pattern of which locations in the memory the client is reading or writing in each protocol execution.[2] The work of the client and server in each such access should be

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

small and bounded by a poly-logarithmic factor in the memory size, where the goal is to access the data without downloading it from the server in its entirely. In stronger attack scenarios, the ORAM is also authenticated which means that the server cannot modify the content of the memory. In particular, the server cannot even “roll-back” to an older version of the data. The efficiency of ORAM constructions is evaluated by their bandwidth blowup, client storage and server storage. Bandwidth blowup is the number of data blocks that are needed to be sent between the parties per request. Client storage is the amount of trusted local memory required for the client to manage the ORAM and server storage is the amount of storage needed at the server to store all data blocks. Since the seminal sequence of works by Goldreich and Ostrovsky, ORAM has been extensively studied [42, 21, 29, 46, 43, 41], optimizing different metrics and parameters.

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

In this work we follow a slightly modified version of the standard definition (of [17, 38, 20]), in which the compiled program P^* is not hardcoded with any secret values, namely, neither secret keys for encryption/authentication algorithms nor the randomness that specifies future memory locations to be accessed by the program, rather, the compiled program obtains these secret values as input. More concretely, P^* is given two inputs: (1) a secret value k that is used to derive the keys for encrypting and authenticating the data, (2) a uniformly random string r which corresponds to the random indices that are accessed during the computation. The formal definition follows:[3]

Definition 2. *A polynomial time algorithm C is an Oblivious RAM (ORAM) compiler with computational overhead $c(\cdot)$ and memory overhead $m(\cdot)$, if C, when given $n \in \mathbb{N}$ and a deterministic RAM program P with memory size n, outputs a program P^* with memory size $m(n) \cdot n$, such that for any input $x \in \{0,1\}^*$, uniformly random key $k \in \{0,1\}^\kappa$ and uniformly random string $r \in \{0,1\}^\kappa$, it follows that $T(P^*(n, x, k, r)) = c(n) \cdot T(P, n, x)$ and there exists a negligible function μ such that the following properties hold:*

- ***Correctness.*** *For any $n \in \mathbb{N}$, any input $x \in \{0,1\}^*$, any key and uniformly random string $k, r \in \{0,1\}^\kappa$, with probability at least $1 - \mu(\kappa)$, $P^*(n, x, k, r) = P(n, x)$.*
- ***Obliviousness.*** *For any two programs P_1, P_2, any $n \in \mathbb{N}$, any two inputs, uniformly random keys and uniformly random strings: $x_1, x_2 \in \{0,1\}^*, k_1, k_2, r_1, r_2 \in \{0,1\}^\kappa$ respectively, if $T(P_1(n, x_1)) = T(P_2(n, x_2))$ and $P_1^* \leftarrow C(n, P_1, \rho_1)$,*

[3] The following definition is derived from the definition given in[8].

$P_2^* \leftarrow C(n, P_2, \rho_2)$ then the access patterns $\mathsf{MemAccess}(P_1^*(n, x_1, k_1, r_1))$ and $\mathsf{MemAccess}(P_2^*(n, x_2, k_2, r_2))$ are computationally indistinguishable, where the random tapes ρ_1, ρ_2 that were used by the compiler to generate the compiled programs are given to the distinguisher. [4]

Note that the above definition (just as the definition of [20]) only requires an oblivious compilation of deterministic programs P. This is without loss of generality: We can always view a randomized program as a deterministic one that receives random coins as part of its input.

Realization of the Modified Definition We present here a sketch of an ORAM compiler that meets the above requirements, which is a slightly modified construction of the Simple ORAM that was presented in [8]. The modified compiler is a deterministic algorithm C, that is, its random tape ρ is an empty string. When given a program P, the compiler outputs a program P^* that takes the inputs x, k, r where x is the input to the original program P, k is a uniformly random string from which the encryption/authentication keys are derived and r is a uniformly random strings of the following form: $r = \{Pos, r_1, r_2, \ldots, r_T\}$ such that Pos is the initial position map of the oblivious program and $r_1, \ldots, r_T$ are the additional random locations that are used for each iteration during the execution of the program P^*. The program P^* that C outputs is specified exactly as the the oblivious program presented in [8], except that the position map Pos and random paths $r_1, \ldots, r_T$ are not hardcoded within the program, rather, they are given as inputs to the program.

2.3 Secure Computation in the RAM Model

We adapt the standard definition for secure two-party computation of [18, Chapter 7] for the RAM model of computation. In this model of computation, the initial input is split between two parties and the parties run a protocol that securely realizes a program P on a pair of "short" inputs x_1, x_2, which are viewed as the initial state of the program. In addition, the program P has random-access to a memory of size n which is initially empty. Using the notations from Section 2.1, we refer to this (potentially random) process by $P^D(x_1, x_2)$. In this work we prove the security of our protocols in the presence of malicious computationally bounded adversaries.

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

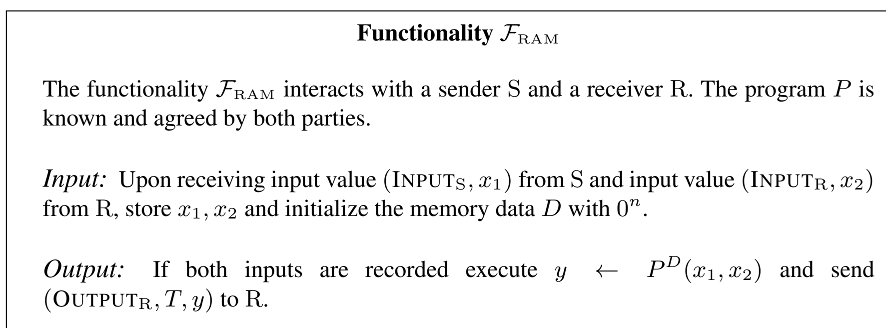
[4] The use of ρ_1, ρ_2 does not reveal any information about the access pattern nor about the encryption key of the data, these are determined only by the keys k_1, k_2 and the random strings r_1, r_2.

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

party throughout the computation of P^D. We present here both definitions, starting with the definition of full security.

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.

Fig. 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.

Definition 3 (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* $i \in \{\mathrm{S}, \mathrm{R}\}$*,*

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

where s *and* κ *are the security parameters.*

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.

The UMA Model In [16], Gentry et al. considered a weaker notion of security, denoted by *Unprotected Memory Access* (UMA), in which the receiver may additionally learn the content of the memory D, as well as the memory access pattern throughout the computation including the locations being read/written and their contents.[6] In the context of two-party computation, when considering the ideal execution, the trusted party further forwards the adversary the values $\mathsf{MemAccess} = \{(i^{\mathsf{read}}_t, i^{\mathsf{write}}_t, b^{\mathsf{write}}_t)\}_{t\in[T]}$ where i^{read}_t is the address to read from, i^{write}_t is the address to write to and b^{write}_t is the bit value to be written to location i^{write}_t 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 4 (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

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

where s *and* κ *are the security parameters.*

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

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

[7] As for RAM programs, ORAM schemes can also support this property. Moreover, the [16] transformation discussed in Section 2.1 can be applied to ORAM schemes as well.

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

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

Input: Upon receiving input value $(\textsc{Input}_{\mathrm{S}}, x_1)$ from S and input value $(\textsc{Input}_{\mathrm{R}}, x_2)$ from R, set $\mathsf{state}_1 = (x_1, x_2)$ 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 number of memory accesses that were performed during the execution, and $\mathsf{MemAccess}$ is the access pattern of the execution.

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

Theorem 2. *Let π be a secure two-party protocol that provides UMA security for RAM programs that support ptWrites in the presence of malicious adversaries and C an ORAM compiler that satisfies ptWrites, then Θ is a two-party protocol that provides full security for arbitrary RAM programs in the presence of malicious adversaries.*

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

Transformation Θ from UMA to Fully Secure Protocol

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

Protocol:

1. The parties generate the randomness ρ using a coin-tossing protocol.
2. The parties agree on the oblivious program $P^* \leftarrow C(P, n, \rho)$ where ρ is C's random tape.
3. The parties run $\pi(P^*)$ where R's inputs are x_1, k_1, r_1, S's inputs are x_2, k_2, r_2 and the program P^* is given n (the memory size), the parties' input $x = x_1 \| x_2$, the key and the random tape of the compiled program, i.e. $k_1 \oplus k_2$ and $r_1 \oplus r_2$, respectively. That is, the parties run $\pi(P^*(n, x_1 \| x_2, k_1 \oplus k_2, r_1 \oplus r_2))$.

Fig. 3. A transformation from UMA to full security.

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

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

1. Let T be the run time of the program P, and let $\tilde{P}$ be the program of the form:
   ```
   For i=0 To T:
       Read(k);
   ```
   for some constant $\texttt{k} \in [n]$ (i.e. k is in the range of D's size). Let $\tilde{P}^* \leftarrow C(n, \tilde{P})$ and let $\mathsf{MemAccess}(\tilde{P}^*, n, \varepsilon)$ be the memory access pattern resulted by its execution (where ε is an empty string, since $\tilde{P}$ gets no input).
2. Given the output of the program $y = P(x, y)$, the simulator $\mathcal{S}_{\text{RAM}}$ outputs the view that is the result of $\mathcal{S}_{\text{UMA}}(y, \mathsf{MemAccess}(\tilde{P}^*, n, \varepsilon))$.

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

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

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

On the Capabilities of Semi-Honest in a Garbled RAM and ORAM Schemes When considering ORAM schemes in the context of two-party computation, it must be ensured that a read operation is carried out correctly. Namely, that the correct element

from the memory is indeed fetched, and that the adversary did not "roll back" to an earlier version of that memory cell. Importantly, this is not just a concern in the presence of malicious adversaries, as a semi-honest adversary may try to execute its (partial) view on inconsistent memory values. Therefore, the scheme must withhold such attacks. Handling the first attack scenario is relatively simply using message authentications codes (MACs), so that a MAC tag is stored in addition to the encrypted data. Handling roll backs is slightly more challenging and is typically done using Merkle trees. In [16] roll backs are prevented by creating a new secret key for each time period. This secret key is used to decrypt a corresponding ciphertext in order to extract the label for the next garbled circuit. By replacing the secret key each time period, the adversary is not able decrypt a ciphertext created in some time period with a secret key that was previously generated.

2.4 Timed IBE [16]

TIBE was introduced by Gentry et al. in [16] 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). We omit the formal definition due to space limitations.

2.5 Garbled RAM Based on IBE [16]

Our starting point is the garbled RAM construction of [16]. Intuitively speaking, garbled RAM [34] is an analogue object of garbled circuits [47, 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 [16], the programs can only be executed in the specified order, where each program obtains a state that depends on prior executions. The [16] garbled RAM proposes a fix to the aforementioned circularity issue raised in [34] 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 $\tilde{D}, \tilde{P}, \tilde{x}$ such that the evaluator reveals the output $\tilde{P}(\tilde{D}, \tilde{x}) = P(D, x)$ and nothing else. A RAM program P with running time T can be evaluated using T copies of a Boolean circuit $\mathrm{C}^P_{\mathrm{CPU}}$ where $\mathrm{C}^t_{\mathrm{CPU}}$ computes the function $\mathrm{C}^P_{\mathrm{CPU}}(\mathsf{state}_t, b^{\mathsf{read}}_t) = (\mathsf{state}_{t+1}, i^{\mathsf{read}}_t, i^{\mathsf{write}}_t, b^{\mathsf{write}}_t)$. Then secure evaluation of P is possible by having the sender S garble the circuits $\{\mathrm{C}^t_{\mathrm{CPU}}\}_{t \in [T]}$ (these are called the garbled program $\tilde{P}$), whereas the receiver R sequentially evaluates these circuits. In order for the evaluation to be secure the state of the program should remain secret when moving from one circuit to another. To this end,

the garbling is done in a way that assigns the output wires of one circuit with the same labels as the input wires of the next circuit. The main challenge here is to preserve the ability to read and write from the memory while preventing the evaluator from learning anything beyond the program's output, including any intermediate value.

The original idea from [34] 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 [16], the proof of [34] does not follow without assuming an extra circularity assumption. In order to avoid circularity, Gentry et al. proposed to replace the PRF with a public-key primitive. As it is insufficient to use a standard public-key cryptosystem (since the circuit must still produce secret keys for each memory location i, storing the keys $\mathsf{sk}_{i,0}, \mathsf{sk}_{i,1}$), the alternative is to use IBE. Below, we briefly describe their scheme.

The read-only solution. The initialized garbled data $\tilde{D}$ contains a secret key $\mathsf{sk}_{i,b}$ in each memory location $i \in [n]$ where $D[i] = b$, such that i, b serves as an identity secret key for the "identity" (i, b). Moreover, each garbled circuit $\mathrm{GC}^t_{\mathrm{CPU}}$ is hardwired with the master public key MPK of an IBE scheme.[8] 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.

In order to write at time step t to memory location $i = i^{\mathsf{write}}_t$ the value $b = b^{\mathsf{write}}_t$, 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

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

problem. Instead, Gentry et al. [16] 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 $\mathrm{GC}^t_{\mathrm{CPU}}$ and is used to write the bit b^{write}_t to memory location i^{write}_t. To do that $\mathrm{GC}^t_{\mathrm{CPU}}$ 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 $\mathrm{GC}^t_{\mathrm{CPU}}$ 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 $\mathrm{C}^t_{\mathrm{CPU}}$, and cannot be determined from either $\mathsf{sk}_{(t,i,b)}$ or the garbled circuit, provided that the TIBE scheme and the garbling schemes are secure.

2.6 Garbled Circuits

The idea of garbled circuit is originated in [47]. Here, a sender can encode a Boolean circuit that computes some PPT function f, in a way that (computationally) hides from the receiver any information but the function's output. In this work we consider a variant of the definition from [16] that abstracts out the security properties of garbled circuits that are needed via the notion of a *garbled circuit with wire labels*. The definition that we propose below stems from the cut-and-choose technique chosen to deal with a malicious sender. Specifically, the sender uses the algorithm Garb to generate s garbled versions of the circuit C, namely $\{\tilde{\mathrm{C}}_i\}_{i \in [s]}$ for some statistical parameter s. Then, in order to evaluate these circuits the sender sends $\{\tilde{\mathrm{C}}_i\}_{i \in [s]}$ along with the garbled inputs $\{\tilde{x}_i\}_{i \in [s]}$, such that $\tilde{x}_i$ is the garbled input for the garbled circuit $\tilde{\mathrm{C}}_i$. The evaluator then chooses a subset $Z \subset s$ and evaluates the garbled circuits indexed by $z \in Z$ using algorithm Eval. Note that in the notion of garbled circuits with wire labels the garbled inputs $\tilde{x}_i$ are associated with a single label per input wire of the circuit $\tilde{\mathrm{C}}_i$; we denote these labels by $\tilde{x}_i = (\mathsf{lbl}^{1,i}_{\mathrm{in},x[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{in}},i}_{\mathrm{in},x[v_{\mathrm{in}}]})$ (where v_{in} is the number of input wires in C and $x = x[1], \ldots, x[v_{\mathrm{in}}]$ is the input to the circuit). The evaluator learns s sets[9] of output-wire labels $\{\tilde{y}_i\}_{i \in [s]}$ corresponding to the output $y = \mathrm{C}(x)$[10], where $\tilde{y}_i = (\mathsf{lbl}^{1,i}_{\mathrm{out},y[1]}, \ldots, \mathsf{lbl}^{v_{\mathrm{out}},i}_{\mathrm{in},y[v_{\mathrm{out}}]})$, but nothing else (for example, it does not learn $\mathsf{lbl}^{1,i}_{\mathrm{out},1-y[1]}$). For clarity, in the following exposition the label $\mathsf{lbl}^{j,i}_{\mathrm{in},b}$ is the label that represents the bit-value $b \in \{0, 1\}$ for the jth input wire ($j \in v_{\mathrm{in}}$) in the ith garbled version of the circuit (for $i \in s$), namely $\tilde{\mathrm{C}}_i$. Analogously, $\mathsf{lbl}^{j,i}_{\mathrm{out},b}$ represents the same, except that it is associated with an output wire (where $j \in v_{\mathrm{out}}$).

We further abstract two important properties of *authenticity* and *input consistency*. Loosely speaking, the authenticity property ensures that a malicious evaluator will not

[9] This s might be different from the s used in the garbling algorithm, still we used the same letter for simplification.

[10] Note that this holds with overwhelming probability since some of the garbled circuits might be malformed.

be able to produce a valid encoding of an incorrect output given the encoding of some input and the garbled circuit. This property is required due to the reusability nature of our construction. Namely, given the output labels of some iteration, the evaluator uses these as the input labels for the next circuit. Therefore, it is important to ensure that it cannot enter an encoding of a different input (obtained as the output from the prior iteration). In the abstraction used in our work, violating authenticity boils down to the ability to generate a set of output labels that correspond to an incorrect output. Next, a natural property that a maliciously secure garbling scheme has to provide is *input consistency*. We formalize this property via a functionality, denoted by $\mathcal{F}_{\rm IC}$. That is, given a set of garbled circuits $\{\tilde{\rm C}_i\}_i$ and a set of garbled inputs $\{\tilde{x}_i\}_i$ along with the randomness r that was used by Garb; the functionality outputs 1 if the s sets of garbled inputs $\{\tilde{x}_i\}_{i=1}^s$ (where $|\tilde{x}_i| = j$) represent the same input value, and 0 otherwise. This functionality is described in Fig. 8 (Appendix A). We now proceed to the formal definition.

Definition 5 (Garbled circuits.). *A circuit garbling scheme with wire labels consists of the following two polynomial-time algorithms:*

- *The garbling algorithm* Garb*:*

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

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

- *The evaluation algorithm* Eval*:*

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

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

Furthermore, the following properties hold.

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

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

where $\left(\{\tilde{\rm C}_i\}_i, \{u, b, \mathsf{lbl}^{u,i}_{{\rm in},b}\}_{u,i,b}\right) \leftarrow \mathsf{Garb}\left(1^\kappa, s, {\rm C}, \{v, b, \mathsf{lbl}^{v,i}_{{\rm out},b}\}_{v,i,b}\right)$ 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 $\left\{\mathsf{lbl}^{1,z}_{\mathrm{out}}, \ldots, \mathsf{lbl}^{v_{\mathrm{out}},z}_{\mathrm{out}}\right\}_{z\in[Z]}, \{v, b, \mathsf{lbl}^{v,z}_{\mathrm{out},b}\}_{v,z\notin[Z],b}$ (i.e. one output label for wires in circuits indexed by $z \notin Z$ and a pair of output labels for wires in circuits indexed by $z \in Z$), we have

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

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

Authenticity. We describe the authenticity game in Figure 7 (Appendix A) where the adversary is obtained a set of garbled circuits and garbled inputs for which the adversary needs to output a valid garbling of an invalid output. Namely, a garbled scheme is said to have *authenticity* if for every circuit C, for every PPT adversary $\mathcal{A}$, every s and for all large enough κ the probability $\Pr[\mathrm{Auth}_{\mathcal{A}}(1^\kappa, s, \mathrm{C}) = 1]$ 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 $\mathcal{F}_{\mathrm{IC}}$ functionality described in Figure 8 (Appendix A).

Realizations of our garbled circuits notion. We require the existence of a protocol Π_{IC} that securely realizes the functionality $\mathcal{F}_{\mathrm{IC}}$ described in Figure 8, in the presence of malicious adversaries. We exemplify this realization using [32] in the full version of the paper.

3 Building Blocks

In this section we show how to overcome the challenges discussed in the introduction and design the first maliciously secure 2PC protocol that does not require duplication of the data and works for every garbling scheme that supports our definition based on wire labels. Recall first that in [16] Gentry et al. have used a primitive called Timed IBE, where the secret-key for every memory location and stored bit (i, b) is enhanced with another parameter: the last time step t in which it has been written to the memory. The secret-key $\mathsf{sk}_{(t,i,b)}$ for identity $\mathsf{id} = (t, i, b)$ is then generated using the hard-coded time secret-key TSK_t. Now, since algorithm KeyGen is randomized, running this algorithm s times will yield s independent secret timed keys. This results in s different values to

be written to memory at the same location, which implies duplication of memory data D. In order to avoid this, our solution forces the s duplicated garbled circuits for time step t to use the same random string r, yielding that all garbled circuits output the same key for the identity (t, i, b). Importantly, this does not mean that we can hard-code r in all those s circuits, since doing this would reveal r when applying the cut-and-choose technique on these garbled circuits as half of the circuits are opened. Clearly, we cannot reveal the randomness to the evaluator since the security definition of IBE (and Timed IBE) does not follow in such a scenario. Instead, we instruct the sender to input the *same randomness* in all s copies of the circuits and then run an input consistency check to these inputs in order to ensure that this is indeed the case. We continue with describing the components we embed in our protocol. An overview of the circuits involved in our protocol can be found in Figure 4 and a high-level overview of the protocol can be found in Section 4.

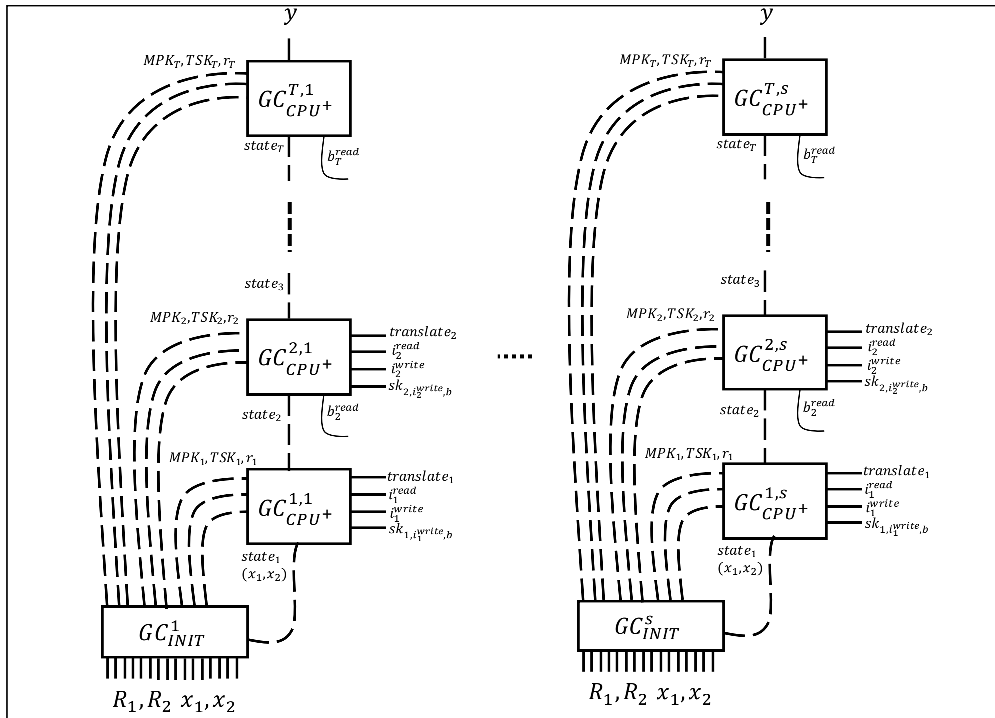


Fig. 4. Garbled chains $\mathrm{GC}_{\mathrm{INIT}}, \mathrm{GC}^{1,i}_{\mathrm{CPU}^+}, \ldots \mathrm{GC}^{T,i}_{\mathrm{CPU}^+}$ for $i \in [s]$. Dashed lines refer to values that are passed privately (as one label per wire) whereas solid lines refer to values that are given in the clear.

3.1 Enhanced CPU-Step Function

The enhanced $\mathsf{cpustep}^+$ function is essentially the CPU-step functionality specified in Section 2.1 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 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 5). We later describe how to securely realize this function and, in particular, how these new additional inputs are generated and given to the T CPU-circuits. The enhanced CPU-step circuit wraps the $\mathsf{WriteTime}$ algorithm defined in Definition 1.

3.2 Initialization Circuit

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

3.3 Batch Single-Choice Cut-And-Choose OT

As a natural task in a cut-and-choose based protocol, we need to carry out cut-and-choose oblivious transfers for all wires in the circuit, for which the receiver picks a subset $Z \subset [s]$ and then obtains either both input labels (for circuits indexed with $z \in Z$), or the input label that matches the receiver's input otherwise. It is crucial that the subset of indices for which the receiver obtains both input labels is the same in all transfers. The goal of this functionality is to ensure the input consistency of the receiver and it is named by "batch single-choice cut-and-choose OT" in [32]. See [**?**] for its formal definition.

In addition to the above, our protocol uses the following building blocks: A garbled scheme $\pi_{\mathrm{GC}} = (\mathsf{Garb}, \mathsf{Eval})$ that preserves the security properties from Definition 5; Timed IBE $\pi_{\mathrm{TIBE}} = (\mathsf{MasterGen}, \mathsf{TimeGen}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and a statistically binding commitment scheme Com.

4 The Complete Protocol

Given the building blocks detailed in Section 3, we are now ready to introduce our complete protocol. Our description incorporates ideas from both [32] and [16]. Specifically, we borrow the cut-and-choose technique and the cut-and-choose OT abstraction

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 $\mathsf{cpu\text{-}step}$ 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_t^{read} and (3) the TIBE related functionalities KeyGen and Enc. Furthermore, the labels lbl_0^{t+1} and lbl_1^{t+1} are hard coded in the circuit.

Inputs.

- state_t - the last state that was output by the previous circuit. We define state_1 to be the parties' inputs x_1, x_2 and set b_0^{read} to be zero.
- b_t^{read} - the last bit that was read from the memory data (i.e. b_t^{read} was read from location i_t^{read}).
- MPK - the master public key of the TIBE scheme.
- TSK_t - a timed secret-key.
- r_t - randomness to be used by algorithms KeyGen and $\mathsf{Enc}_{\mathsf{MPK}}$.

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

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

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

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

Finally, the circuit computes $\mathsf{sk}_{(t,i,b)} = \mathsf{KeyGen}(\mathsf{TSK}_t, \mathsf{id} = (t, i_{t+1}^{\mathsf{write}}, b_{t+1}^{\mathsf{write}}))$ and outputs

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

Fig. 5. The CPU-step circuit.

from [32] (where the latter tool enables to ensure input consistency for the *receiver*). Moreover, we extend the garbled RAM ideas presented in [16] for a maliciously secure two-party protocol in the sense that we modify their garbled RAM to support the cut-and-choose approach. This allows us to obtain constant round overhead. Before we delve into the details of the protocol, let us present an overview of its main steps:

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

Initialization Circuit $\mathrm{C}_{\mathrm{INIT}}$

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

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)

- $r^{\mathrm{MasterGen}}$ used to generate the keys MPK and MSK.
- $r_1^{\mathrm{TimedGen}}, \ldots, r_T^{\mathrm{TimedGen}}$ used to generate the timed secret-keys $\mathsf{TSK}_1, \ldots, \mathsf{TSK}_T$.
- $r_1^{\mathrm{KeyGen}}, \ldots, r_T^{\mathrm{KeyGen}}$ used to generate secret-keys $\{\mathsf{sk}_{t,i,b}\}_{t \in [T], i \in [n], b \in \{0,1\}}$ written to memory.
- $\{r_{t,b}^{\mathrm{Enc}}\}_{t \in [T], b \in \{0,1\}}$ are used by the encryption algorithm within the CPU circuits. (Recall that the tth enhanced CPU step circuit $\mathrm{C}^t_{\mathrm{CPU}^+}$ encrypts the two labels of the input wire that corresponds to the input bit of the next circuit $\mathrm{C}^{t+1}_{\mathrm{CPU}^+}$.)

Then, the circuit computes:

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

Outputs.

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

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

Fig. 6. Initialization Circuit $\mathrm{C}_{\mathrm{INIT}}$.

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

- **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 2.6.
- **Evaluation.** Upon verifying their consistency, R uses the input labels and evaluates all evaluation chains, such that in every time step t it discards the chains that their outputs $(i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, \mathsf{sk}_t, \mathsf{translate}_t)$ do not comply to the majority of the outputs in all evaluation chains. We put a spotlight on the importance of the random strings R_1, R_2 that the parties provide to the chains, these allow our protocol to use a *single* block of data D for *all* threads of evaluation, which could not be done in a trivial plugging of the cut-and-choose technique. As explained in Definition 5, verifying the correctness of the check chains can be done given only (both of the) input labels for C_{INIT} circuits.

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

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

   for $v \in [v_{\mathrm{out}}], i \in [s], b \in \{0,1\}$ and r^t_g the randomness used within Garb.
   – Interpret the result labels $\{\mathsf{lbl}^{u,i,t}_{\mathrm{in},b}\}_{u,i,b}$ as the following groups of values: $\mathsf{state}_t, b^{\mathsf{read}}_t, \mathsf{MPK}_t, \mathsf{TSK}_t$ 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 $\mathrm{C}^t_{\mathrm{CPU}^+}$.
   – Choose random labels for the output wires correspond to $i^{\mathsf{read}}_t, i^{\mathsf{write}}_t, \mathsf{sk}_{t,i,b}$ 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 $\{\mathsf{lbl}^{v,i,t}_{\mathrm{out},b}\}_{v,i,b}$.
   – Garble $\mathrm{C}^t_{\mathrm{CPU}^+}$ by calling

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

   with $\{\mathsf{lbl}^{v,i,t}_{\mathrm{out},b}\}_{v,i,b}$ the set of labels from above and r^t_g the randomness used within Garb.
   – Interpret the result labels $\{\mathsf{lbl}^{u,i,t}_{\mathrm{in},b}\}_{u,i,b}$ as the following groups of values: $\mathsf{state}_t, b^{\mathsf{read}}_t, \mathsf{MPK}_t, \mathsf{TSK}_t$ 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 $\mathrm{C}_{\mathrm{INIT}}$:
   – 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 $\{\mathsf{lbl}^{v,i}_{\mathrm{out},b}\}_{v,i,b} = \big\{\mathsf{lbl}^{u,i,1}_{\mathsf{state},b} \cup \mathsf{lbl}^{u,i,t}_{\mathsf{MPK}_t,b} \cup \mathsf{lbl}^{u,i,t}_{\mathsf{TSK}t,b} \cup \mathsf{lbl}^{u,i,t}_{r_t,b}\big\}_{u,i,b}$ for all u, i, t, 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 $\{\mathsf{lbl}^{u,i}_{\mathrm{S},b}\}_{u,i,b}$ 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 $\mathbf{v_1}, \ldots, \mathbf{v}_\ell$ so that $\mathbf{v_j}$ contains the s pairs of random labels associated with R's jth input bit $x_2[j]$ in all garbled circuits $\mathrm{GC}^1_{\text{INIT}}, \ldots, \mathrm{GC}^s_{\text{INIT}}$.
   (b) R inputs a random subset $Z \subset [s]$ of size exactly $s/2$ and bits $x_2[1], \ldots, x_2[\ell]$.
   (c) The result of Π_{SCCOT} is that R receives *all* the labels associated with its input wires in all circuits $\mathrm{GC}^z_{\text{INIT}}$ for $z \in Z$, and receives a single label for every wire associated with its input x_2 in all other circuits $\mathrm{GC}^z_{\text{INIT}}$ for $z \notin Z$.
3. SEND GARBLED CIRCUITS AND COMMITMENTS.
   S sends R the garbled circuits chains $\mathrm{GC}^i_{\text{INIT}}, \mathrm{GC}^{1,i}_{\text{CPU}^+}, \ldots, \mathrm{GC}^{T,i}_{\text{CPU}^+}$ 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 $\mathrm{GC}_{\text{INIT}}$.
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 $\mathrm{GC}^z_{\text{INIT}}$ for every $z \in Z$. If the values received by S are incorrect, it outputs $\perp$ and aborts. Chains $\mathrm{GC}^z_{\text{INIT}}, \mathrm{GC}^{1,z}_{\text{CPU}^+}, \ldots, \mathrm{GC}^{t,z}_{\text{CPU}^+}$ 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 $\mathrm{GC}^z_{\text{INIT}}$ (from Steps 2 and 5). This means that it can check the correctness of the chains $\mathrm{GC}^z_{\text{INIT}}, \mathrm{GC}^{1,z}_{\text{CPU}^+}, \ldots, \mathrm{GC}^{T,z}_{\text{CPU}^+}$ 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 $\big\{(\mathsf{lbl}^{1,z}_{\text{in},x_1[1]}, \ldots, \mathsf{lbl}^{\ell,z}_{\text{in},x_1[\ell]})\big\}_{z \notin [Z]}$ for its input x_1.
   - S and R participate in the input consistency check protocol Π_{IC}.
     - The common inputs for this protocol are the circuit C_{INIT}, its garbled versions $\{\mathrm{GC}^i_{\text{INIT}}\}_{z \notin Z}$ and the labels $\big\{(\mathsf{lbl}^{1,z}_{\text{in},x_1[1]}, \ldots, \mathsf{lbl}^{\ell,z}_{\text{in},x_1[\ell]})\big\}_{z \notin [Z]}$ that were sent before.
     - S inputs its randomness r^0_g and the set of output labels $\{\mathsf{lbl}^{v,i}_{\text{out},b}\}_{v,i,b}$ that were used within Garb on input $\mathrm{GC}_{\text{INIT}}$, 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 $\mathrm{GC}^z_{\text{INIT}}$ using Eval and the input wires it obtained in Step 7 and reveal one label for each of its output wires $\mathsf{lbl}^{\text{out},z}_{\text{INIT}}$.
   (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[11]. 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 $\tilde{Z} = \tilde{Z} \setminus \{z' \mid \mathsf{out}_{t,z'} \neq \mathsf{out}_t\}$. This means that R halts the execution thread of the circuit copies that were found flawed during the evaluation.

iii. Output out_{T+1}.

Theorem 3. *Assume that π_{GC} is a garbling scheme (cf. Definition 5), that π_{TIBE} is TIBE scheme and that* Com *is a statistical binding commitment scheme. Then, protocol Π^P_{UMA} securely realizes $\mathcal{F}_{\mathrm{UMA}}$ in the presence of malicious adversaries in the $\{\mathcal{F}_{\mathrm{SCCOT}}, \mathcal{F}_{\mathrm{IC}}\}$-hybrid for all program executions with ptWrites.*

High-level overview of our proof. In this section we present the intuition of why does our protocol secure in the UMA model, while a full proof of lemma 3 presented in the full version of the paper. 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 [32], the selective-bit attack cannot be carried out successfully since R obtains all the input keys associated with its input in the cut-and-choose oblivious transfer, where the labels associated with both the check and evaluation circuits are obtained together. Thus, if S attempts to run a similar attack for a small number of circuits then it will not effect the majority, whereas if it does so for a large number of circuits then it will be caught with overwhelming probability. In the protocol, R checks that half of the chains and their corresponding input garbled values were correctly generated. It is therefore assured that with high probability the majority of the remaining circuits and their input garbled values are correct as well. Consequently, the result output by the majority of the remaining circuits must be correct.

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

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

table translate that correctly encrypts the active label (namely, the label observed by the adversary), yet encrypts a fake inactive label. We then show, that the real view in which all labels are correctly encrypted, is indistinguishable from the simulated view in which only the active label is encrypted correctly.

References

1. Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In *EUROCRYPT*, pages 702–729, 2015.
2. Donald Beaver. Foundations of secure interactive computing. In *CRYPTO*, pages 377–391, 1991.
3. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In Harriet Ortiz, editor, *22nd STOC*, pages 503–513. ACM, 1990.
4. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *CCS*, pages 784–796, 2012.
5. Dan Boneh and Xavier Boyen. Efficient selective identity-based encryption without random oracles. *J. Cryptology*, 24(4):659–693, 2011.
6. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003.
7. Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.
8. Kai-Min Chung and Rafael Pass. A simple oram. Cryptology ePrint Archive, Report 2013/243, 2013. http://eprint.iacr.org/2013/243.
9. Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, pages 73–80, 1972.
10. Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.
11. Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty ram computation in constant rounds. To appear in TCC, 2016.
12. Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. 2015.
13. Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *STOC*, pages 449–458, 2015.
14. Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PETS*, pages 1–18, 2013.
15. Craig Gentry, Shai Halevi, Charanjit S. Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. *IACR Cryptology ePrint Archive*, 2014:345, 2014.
16. Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, pages 405–422, 2014.
17. Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987.
18. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.
19. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.
20. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

21. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, pages 157–167, 2012.
22. S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, pages 513–524, 2012.
23. Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In *CRYPTO*, pages 150–169, 2015.
24. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In *EUROCRYPT*, pages 406–425, 2011.
25. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, pages 572–591, 2008.
26. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In *TCC*, pages 294–314, 2009.
27. Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114, 2007.
28. Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *ASIACRYPT*, pages 506–525, 2014.
29. Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156, 2012.
30. Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO (2)*, pages 1–17, 2013.
31. Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, pages 52–78, 2007.
32. Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *TCC*, pages 329–346, 2011.
33. Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient ram-model secure computation. In *IEEE Symposium on Security and Privacy*, pages 623–638, 2014.
34. Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, pages 719–734, 2013.
35. Peihan Miao. Cut-and-choose for garbled ram. Manuscript.
36. Silvio Micali and Phillip Rogaway. Secure computation (abstract). In *CRYPTO*, pages 392–404, 1991.
37. Jesper Buus Nielsen and Claudio Orlandi. Lego for two-party secure computation. In *TCC*, pages 368–386, 2009.
38. Rafail Ostrovsky. Efficient computation on oblivious rams. In *STOC*, pages 514–523, 1990.
39. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, pages 250–267, 2009.
40. Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.
41. Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX*, pages 415–430, 2015.
42. Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with o((logn)3) worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.
43. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, pages 299–310, 2013.
44. Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In *CCS*, pages 850–861, 2015.

45. Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *CCS*, pages 191–202, 2014.
46. Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *CCS*, pages 293–304, 2012.
47. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.
48. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

A Garbled Circuits

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:

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

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

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

The definition of garbled circuits with respect to the cut-and-choose technique is presented in Section 2.6. In this section we present the Input Consistency Functionality (Figure 8) 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 $\left\{(\mathsf{lbl}^{1,i}_{\text{in},x[1]},\ldots,\mathsf{lbl}^{v_{\text{in}},i}_{\text{in},x[v_{\text{in}}]})\right\}_{i\in[s]}$.
- 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

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

  - If $b_i = \bot$ for some i then output 0. Also If $b_1\neq b_i$ for some i output 0. (This checks that all labels are interpreted as the same input bit in all garbled circuits).
- Given that the above algorithm has not output 0, then output 1.

Fig. 8. The input consistency functionality $\mathcal{F}_{\text{IC}}$.