Adaptively Secure Computation for RAM Programs

Laasya Bangalore[1], Rafail Ostrovsky[2], Oxana Poburinnaya[3], and Muthuramakrishnan Venkitasubramaniam[1]

[1] Georgetown University {lb1264, mv783}@georgetown.edu
[2] UCLA rafail@cs.ucla.edu
[3] Ligero, Inc. oxanapob@bu.edu

Abstract. In this work, we study the communication complexity of secure multiparty computation under minimal assumptions in the presence of an adversary that can adaptively corrupt all parties eventually. Specifically, we are interested in understanding the complexity when modeling the underlying function as a RAM program. Under minimal assumptions, the work of Canetti et al. (STOC 2017) and Benhamouda et al. (TCC 2018) give protocols whose communication complexity grows quadratically in the circuit size when the computation is expressed as a Boolean circuit. In this work, we obtain the first two-round two-party computation protocol, which is secure against adaptive adversaries who can adaptively corrupt all parties where the communication complexity is proportional to the square of the RAM complexity of the function up to polylogarithmic factors assuming the existence of non-committing encryption.

Keywords: Adaptive Security · Garbled RAM · Secure Computation · Oblivious RAM

1 Introduction

Introduced by Yao [31] and Goldreich, Micali, and Wigderson [21], secure multiparty computation (MPC) allows a set of mutually distrustful agents to collaborate and accomplish a common goal while preserving each agent's privacy to a maximal extent. Secure computation enables anonymous electronic elections, privacy-preserving electronic auctions (or contract biddings), privacy-preserving data mining, fault-tolerant distributed computations, and more.

Typically, an adversary is modeled as a single computational entity that has the capability of hacking, or corrupting, an arbitrary subset of the communicating parties over a network and launching a coordinated attack. The classical and most popular model assumes that the set of corrupted parties are compromised before the target protocol begins. This is referred to as the *static* corruption model. The *adaptive* corruption model, introduced by Canetti et al. [5], considers a stronger adversary that can hijack a host at *any* time during the course of the computation. Arguably, adaptive security provides more meaningful security. In particular, it captures "hacking" attacks where an external attacker breaks into parties' machines in the midst of a protocol execution, and can choose its targets adaptively based on the currently available information. Furthermore, an interesting side-effect of adaptive security is that in many cases it automatically guarantees some form of resilience against leakage from side-channel attacks [3,4].

In this work we focus on *full* adaptive security, namely security against an adversary who can (eventually) corrupt *all participants*. Achieving full adaptive security offers strong protocol compositional features, namely, embedding a full adaptive secure sub-protocol in a larger system enables arguing security of the larger system in a modular way even when all participants in the sub-protocol are corrupted. In the adaptive setting, the case when everybody can be corrupted is usually the most difficult to deal with; this should be contrasted with the static setting, where security in such a case comes for free[4].

Canetti et al. [7] established the feasibility of fully adaptively secure protocol where they designed an $O(d)$-round protocol to securely realize any functionality. In the static setting, constant-round protocols were known [26,22] and the gap in round complexity between what was feasible in the static and fully adaptive regime remained an open problem. When assuming reliable erasures, a constant-round protocol was given by Garg and Sahai [19]. The gap was closed simultaneously in the works of by [18,14,6] where they designed the first constant round protocol. These protocol assumed the trusted generation of a common reference string and relied on the strong assumption of existence of indistinguishability obfuscation. Moreover, these construction required a common reference string proportional to size of the circuit. More recently, Canetti et al. in [9] provided the first constant-round fully adaptively secure protocol from standard (minimal) assumptions. The question of the precise round complexity was finally resolved in the work of [2], where they constructed a two-round fully adaptive protocol. In essence, these works closed the gap in round complexity between static and fully adaptively secure protocols.

In this work, we are interested in understanding the communication complexity of fully adaptive protocols in the constant-round regime under minimal assumptions. The works of [18,14,8] achieve essentially an optimal communication complexity where it is independent of the circuit size. However, as mentioned above, they rely on strong assumptions and the trusted sampling of a common reference string that is as large as the circuit being computed. Moreover, a bound on the size of the circuit to be securely computed was required at the time of the CRS generation. Another vein of works [8,11] show how the complexity can be made proportional to the RAM complexity as opposed to the circuit complexity. The protocol of Cohen, shelat and Wichs [12] improve these works to achieve communication and the CRS size that is only proportional to the *depth* of the circuit.

Under minimal assumptions, we only have the work of Canetti et al. [9] and Benhamouda et al. [2] and their communication complexity grows quadrati-

[4] Indeed, in this case the static simulator obtains the inputs of all parties in the protocol and thus can simulate the execution by simply running the protocol.

cally in the circuit size when the circuit is expressed as a boolean circuit. The main question that is left open is to understand the communication complexity under minimial assumptions, namely:

What is the communication complexity of constant-round fully adaptive MPC protocols?

In this work, we make progress in answering this question. Specifically, we show how to obtain better complexity that [9] when the circuit is expressed as a RAM computation.

In the static case, RAM-efficient protocols have been obtained in the plain model via *garbled RAM*, a primitive introduced by Lu and Ostrovsky in 2013 [28]. This primitive allows to separately garble the memory, the input, and the RAM program (without converting it into a circuit), such that the size and runtime of the garbled RAM program is only proportional to the runtime of the program, up to polylogarithmic factors. The original paper required a strong circular-security assumption, but in sequence of follow-up works [28,20,17,16] the assumption was improved to a black-box use of any one-way function while maintaining poly-logarithmic overhead in all parameters. Equipped with garbled RAM, several recent works have demonstrated constant round MPC protocols with communication proportional to the RAM complexity in the static setting [24,15].

However, the state of affairs in the adaptive setting leaves us with either a construction proportional to the boolean circuit complexity under minimal assumptions [9] or relying on strong assumptions with a huge CRS [14,6,8,12].

Therefore, the main question that we ask in this paper is this:

Can we construct constant-round secure computation for RAM programs with only poly-logarithmic overhead that withstands (full) adaptive corruptions in the plain model?

In this work, we make a significant step towards answering this question in the affirmtive where we provide a construction whose communication complexity is proportional to the square of the RAM complexity (upto polylogarithmic factors). The question of whether we can reduce the quadratic overhead to linear remains an intriguing one. We remark that even in the case of circuits the best construction we have so far is quadratic in circuit size. To address the malicious case, we additionally design the first RAM-efficient zero-knowledge proof system that is adaptively secure [7,27,23]. Previously such constructions were known only for circuits and required non-constant number of rounds [23].

1.1 Our Results

In this paper, we provide the first construction of a secure two-party computation protocol for RAM programs that withstands adaptive corruption of both parties by an active adversary.

Our first result is an ORAM compiler that is adaptively secure. Informally, we say that an ORAM compiler is adaptively secure if there exists algorithms Sim_1 and Sim_2 such that Sim_1 given memory size and running time as inputs can provide the memory access sequence along with some state information, and Sim_2 given state and the actual input x as inputs can output the randomness for the compiler that leads to the simulated memory access sequence. We have the following theorem:

Theorem 1 (Informal). *There exists an adaptively secure ORAM with* $\mathsf{polylog}(n)$ *worst-case computational overhead and* $\mathsf{polylog}(n)$ *memory overhead, where n is the memory size.*

Next, we construct a functionally equivocal encryption scheme that is RAM-efficient, which we combine with an adaptively-secure ORAM to obtain our main theorem.

Theorem 2 (Informal). *Assume existence of two-round oblivious transfer secure against passive corruption of both parties by an adaptive adversary then there exists:*

– *A* minimum interaction *(i.e., two-message) two-party general function evaluation protocol for functionalities expressed via a RAM program Π that withstands passive corruption of both parties by an adaptive adversary. The protocol does not use data erasures.*
– *If Π's running time is T, the sum of the input sizes of the parties is n and the size of the memory accessed by Π is M, then our communication complexity is $\widetilde{O}((M+n+T)^2)$. Here $\widetilde{O}(\cdot)$ ignores* $\mathsf{poly}(\log T, \log n, \kappa)$ *factors where κ is the (computational) security parameter.*

Noting that the required oblivious transfer protocol from the theorem can be constructed based on any non-committing encryption scheme [5], we obtain the first constant-round adaptively secure two-party computation for RAM programs in the plain model based on standard assumptions.

Finally, we design a (RAM-efficient) adaptive zero-knowledge proof in the UC-model whose communication complexity is $\widetilde{O}((M+n+T)^2)$ which we combine with our semi-honest protocol to obtain an adaptively secure 2PC computation that secure against malicious adversaries.[5] Formally, we have:

Theorem 3 (Informal). *Assuming collision resistant hashing and dense cryptosystems, there exists a constant-round* UC-secure *two-party general function evaluation protocol in the common random string model, in face of active corruption of all parties by an adaptive adversary, where communication complexity is $\widetilde{O}((M+n+T)^2)$ and the length of the common random string is independent of the size of the function.*

[5] We remark that to obtain our result, it sufficient to obtain a circuit-efficient adaptive zero-knowledge proof.

1.2 Our Techniques

The Challenge of Adaptive Security. When considering adaptive secure computation, the best construction for circuits under minimal assumptions has communication complexity $O(s^2\mathsf{poly}(\kappa))$ for a circuit of s gates. As mentioned before, a long line of research in MPC has focused on measuring the complexity of constructions w.r.t the RAM complexity of the underlying function and designed protocols whose complexity are proportional to the RAM complexity as opposed to the circuit complexity. We recall here that the transforming a RAM program to a circuit is prohibitive where the best constructions that start from a T-time program compile into a circuit of size $O(T^3 \log T)$ [13,29]. Hence, converting a RAM program to a circuit and relying on the construction by [9] would result in a construction of complexity $O(T^6\mathsf{poly}(\log T, \kappa))$. Following analogous works in the static setting, our approach is to design a construction directly for RAM programs.

In a nutshell, our approach extends the equivocal garbling scheme of [9] to garbled RAM. As we describe next, this will not be a simple combination of techniques and we need to overcome a few obstacles. Looking ahead, we will not be able to extend any arbitrary Garbled RAM construction and make it equivocal. This gives rise to three key obstacles.

Obstacle 1: Deterministic vs Randomized functionality. All Garbled RAM constructions rely on a pre-processing step where the memory sequence is made oblivious, typically by applying an Oblivious RAM (ORAM) compiler. This means that even when the underlying function that we wish to securely evaluate is deterministic, the effective functionality we garble will be randomized. In the adaptive case, randomized functionalities are tricky and they are impossible in the general case [25] at least in the plain model. In the CRS model, we do have constructions [14,6,8,12] based on indistinguishability obfuscation. We resolve this by considering the specific randomization used in the construction and show that it is "equivocable". More precisely, we will need an Oblivious RAM (ORAM) compiler to be adaptively secure. In other words, we need an ORAM scheme where the simulator first provides a sequence of memory accesses before knowing actual program inputs and, later, after learning the inputs, output randomness for the ORAM compiler that maps the actual memory sequence to the simulated one. As our first contribution, we provide the first construction of an ORAM scheme with this adaptive property.

Obstacle 2: Equivocating memory. Performing memory read and write operations is a challenging component of Garbled RAM constructions as the memory locations to be read are only known during *run* time. In order to incorporate data read from memory into the computation, the labels corresponding to the data need to be provided as input to the garbled step circuits [6]. A common

[6] In a RAM program, a step circuit performs CPU computations at a particular time step.

way of doing this is to encrypt the labels under some key and provide a mechanism for determining these keys during the run-time, once the data location to be read is known. Prior works [17,20] incorporate ways to efficiently generate these keys needed to encrypt the labels for the next circuit. In [20], a master key is used to generate keys corresponding to the memory location to be read and then this key is used to encrypt the labels associated with the read bits. In [17], a tree-based structure is used to pick the appropriate key to encrypt the labels associated with the data. At a very high-level, their garbled memory has a tree-structure with the encrypted memory at the leaf nodes. Their construction navigates through the tree towards the data to be read and obtains the appropriate key to encrypt the label (for the next step circuit). A common thread between these techniques is that they obtain efficiency by compressing the keys, either by using a master key to generate other keys or using a tree-structure of keys. But such a compression makes it hard for the simulator to "equivocate" the keys. Consequently, we will not be able to rely on the Garbled RAM constructions of [20,17].

Obstacle 3: RAM-efficient Equivocal Encryption. A crucial ingredient in [9] is a Functionally Equivocal Encryption (FEE) scheme that allows a private-key encryption scheme to be equivocated with a key size smaller than the message length when the equivocation space can be expressed as the image of a function over a smaller domain. Specifically, [9] provides an FEE scheme for functions expressed as a circuit where the ciphertext size is proportional to the circuit size and the keys are proportional to the input size of the function (rather than message length being encrypted). As Garbled RAM construction employs sequence of circuits that are typically garbled, one approach is to use an FEE scheme to garble these circuits. The issue here is that the function f that defines the message space is a RAM program and relying on the FEE scheme from [9] that is constructed for circuits will be inefficient. The main challenge here is to construct a variant of the equivocal encryption that is efficient for RAM programs. Following the blueprint of [9], we can convert a Garbled RAM construction to an encryption scheme where the key size is small, the main challenge however is to ensure that one can equivocate the randomness consistent with the encryption algorithm. We show how the Garbled RAM scheme of [17] can be converted to a RAM-efficient equivocal scheme.

Obstacle 4: Obtaining Malicious Security Extending our result to obtain malicious security requires RAM-efficient adaptive zero-knowledge which was previously known only based on indistinguishability obfuscation [18,9]. We obtain malicious security by combining our protocol with an adaptive zero-knowledge proof [7,27,23] using the classic GMW compiler. In order to maintain the communication complexity we need a zero-knowledge proof whose complexity is linear in the circuit size of the NP-relation. The work of [23] provides such a construction, however, it requires a non-constant number of rounds. In fact, the round complexity is proportional to circuit size and will not be sufficient to get our result. We address this by designing a more efficient proof system. In fact, for any NP-relation expressed as a RAM program we design an adaptive zero-knowledge proof with polylogarithmic overhead.

On Database Size. We remark that the database size influences the *online* communication complexity (i.e. the size of the garbled input) of our equivocal garbled RAM. For a database of size M, our equivocal garbled RAM has an *online* communication complexity of $\widetilde{O}((M + n) \cdot n)$. When we consider a scenario $M >> T$, as is typical in RAM applications, our garbling scheme's online complexity will not be efficient (in fact, it is bigger than the size of the computation). On the other hand, if $M << T$ (or even empty) then our garbled RAM scheme will be online efficient. We emphasize here that our main goal is to design a secure two-party computation protocol with a desired communication complexity and the equivocal garbled RAM scheme is only a means to this goal. In other words, even though the stand-alone primitive garbled RAM scheme is not efficient in all regimes, in the context of our 2PC protocol it suffices to consider $M << T$.

Corruption-adaptive vs. Input-adaptive garbling. We note that the term "adaptive" is also used in the literature [1] to denote a very different property which we will call "input-adaptive security", to distinguish it from "corruption-adaptive security". We only focus on achieving "corruption-adaptive security" which we simply refer to as "adaptive security" in this paper. "Corruption-adaptive security" says that the garbling should remain secure even if the adversary prefers to corrupt the evaluator first and sees the communication (i.e. the garbled circuit and garbled input), and later corrupt the garbler and see its internal state (i.e. randomness used to garble the circuit). Whereas the "input-adaptive security" guarantees security against adversaries that can adaptively choose the input to the circuit even after seeing the garbled circuit. In their setting, the input to the circuit may not be fixed when the circuit is being garbled. [1] considers the garbling process to have two phases: the circuit garbling phase followed by the input garbling phase. In particular:

- Input-adaptive garbled circuits remain secure when the input to the computation is chosen adaptively, but lose security properties if randomness of the garbling is revealed to the adversary.
- Corruption-adaptive garbled circuits remain secure when the randomness is revealed to the adversary, but lose security for adaptively chosen inputs.

Outline. For the lack of space, we defer the definitions to the full version; In section 2, we provide an equivocal ORAM compiler. Section 3 contains the description of RAM-efficient Equivocal Encryption. We give our constructions of Equivocal Garbling and adaptive ZK for RAM in sections 4 & 5 respectively.

2 Equivocal ORAM

In this section, we prove that the ORAM construction of [10] is adaptively secure. We begin by providing an overview of the ORAM Compiler given by

[10]. Consider a client-server setting where a secure client runs a program P that accesses memory D that belongs to an untrusted server. Informally, the ORAM compiler ensures that the server does not learn anything about the client's computation by examining the memory access pattern (*Obliviousness*) while still learning the output of the execution of the RAM program (*Correctness*)[7].

Data Representation. Let the server's memory D, which is of size n, be divided into n/α blocks where the size of each block is α (for some $\alpha > 0$). For any program P with memory of size n, we maintain two main data structures, one at the client's end and the other at the server's end.

The server maintains a complete binary tree of depth $d = \lfloor \log(\frac{n}{\alpha}) \rfloor$. A block is said to be *associated* with a leaf node if the block is stored in one of the nodes on the path from the root to that leaf node. Each node can store at most k tuples of the form (b, l, val), where b denotes the block number, l denotes the leaf node and val denotes the content of block b. If any node has more than k tuples, then we say an overflow has occurred.

The client maintains an array of size n/α, denoted by Pos. This array maps blocks to leaf nodes. More specifically, the i^{th} position of the array i.e. $Pos[i]$, corresponds to the i^{th} block and stores the leaf node *associated* with this block.

Construction. Given a program P with memory D, the ORAM compiler C outputs a program P' with memory D'. Suppose the execution $P^D(x)$ performs m memory access operations. Let the i^{th} operation be represented by $Op(i, addr, val)$ where $addr \in [n]$ denotes the memory address and val denotes the value at memory address $addr$. Also, let $b := addr/n$ be the block that contains the memory cell $addr$ and $r := addr \ mod(n)$ be the relative position of the memory cell within the block b. After the ORAM compilation, the program P' makes $2m$ memory accesses to D', which are denoted by $\{l_1, ..., l_{2m}\}$[8]. The compiler C replaces each memory access $Op(i, addr, val)$ made by P with the following three steps:

1. **Fetch**: If $Pos[b] = \perp$, then set l_{2i} to a randomly sampled leaf node; Otherwise set $l_{2i} := Pos[b]$ i.e. the leaf node associated with block b. Next, traverse the path from the root to the leaf node l_{2i}, reading and writing back the contents of each of the nodes on this path. If any of the nodes contains a tuple of the form (b, l_{2i}, v), then erase this tuple; otherwise set $v = \perp$. Output the value at position r in v.
2. **Update**: Choose a leaf node $l^\star$ uniformly at random and set $Pos[b] := l^\star$. If $val = \perp$ (i.e. it's a write operation), then update val to v. Add the tuple $(b, l^\star, val)$ to the root of the tree. Abort, if an overflow occurs.
3. **Evict**: Choose another leaf node r uniformly at random and set $l_{2i+1} := r$. Traverse each node on the path from the root to leaf node l_{2i+1} such that:

[7] We adopt the definition of ORAM from [10].

[8] Note that for simplicity we only specify the leaf nodes accessed by $P'^{D'}(x)$ instead of specifying each node accessed along the path from the root to the leaf

```
                              Sim_1(m, n)

1. Choose $2m$ leaf nodes randomly from $[n/\alpha]$. Let $l_i^{sim} \leftarrow [n/\alpha]$ for each
   $i \in [1, 2m]$
2. Output $\{l_1^{sim}, ..., l_{2m}^{sim}\}$
```

Fig. 1. Description of Simulator Sim_1, which outputs the memory access pattern of an ORAM

every tuple (b', l', v') is pushed down along the path towards l_{2i+1} as long as it is still on the path associated with its leaf node l'. Abort, if an overflow occurs at any of the nodes.

Recursion. In the above construction, the client's memory size is $O(n)$ since it needs to store the position map of size n/α. Instead of storing the position map directly, it can be simulated using the ORAM compiler C. So, any read or write operations to the position map will be performed as described in the construction of C. Now, the client only needs to store a new position of size n/α^2. This reduction in the size of the position map can be done recursively until the size is reduced to just $O(k)$ i.e. constant.

Theorem 4. *[10] The ORAM compiler C described above is adaptively secure and has worst-case computational complexity of $O(n \cdot \mathsf{polylog}(n))$ and memory complexity of $O(n \cdot \mathsf{polylog}(n))$ where n is the size of the memory.*

Proof. Correctness follows directly from the construction as $P'^{D'}(x)$ has the same output as $P^D(x)$ for any deterministic function P, memory D and input x, whenever overflow does not occur. Using the same argument as [10], it can be shown that overflows occurs with negligible probability.

Obliviousness on the other hand follows trivially from our simulation Sim_1 as it simply chooses two independent random leaf nodes to be traversed for each memory access operation.

Adaptive security can be proved by showing that the adversary can first see the sequence of memory accesses without knowing the input and later gain access to the internal randomness that is consistent with the input (as well as the memory accesses seen earlier). Assume that an overflow does not occur. We first show that the sequence of memory accesses made by the program P' can be generated without knowing the input x. This can be done using the algorithm Sim_1 which chooses a sequence of random memory locations given just the memory size n and the number of memory accesses m (described in figure 1). Next, given a fixed sequence of memory locations $\overrightarrow{M}$, memory D and input x, we need to show that there exists randomness that accesses memory locations $\overrightarrow{M}$. That is, we need a way of generating randomness r_{eq} such that the memory accesses made by $P'^{D'}(x)$ exactly correspond to $\overrightarrow{M}$, described by

$\mathsf{Sim}_2[\overrightarrow{M}, P](D, x)$

Parameters. P is the RAM that runs in T steps. Let $\overrightarrow{M}$ parsed as $\{l_1^{sim}, ..., l_{2T}^{sim}\}$ denote the memory accesses made by the program after ORAM compilation[a].

1. If $|\overrightarrow{M}| \neq 2T$, then *abort*. Parse $\overrightarrow{M}$ as $(l_0^{sim}, ..., l_{2T-1}^{sim})$ where $(l_\tau^{sim}, l_{\tau+1}^{sim})$ are the two leaf nodes accessed in the τ^{th} time step.
2. Compute the original memory addresses $\overrightarrow{M}_{orig}$ accessed by program P given input x and data D. Parse $\overrightarrow{M}_{orig}$ as $(addr_1, ..., addr_T)$
3. For every memory access operation $\tau \in [T]$, r_τ^{sim} is the randomness used to update the leaf node associated with the fetched block (refer to the *Update* procedure described in ORAM construction). The goal is to compute r_τ^{sim} for each time step τ given $\overrightarrow{M}_{orig}$ and $\overrightarrow{M}$.
4. If there exists a $\tau' > \tau$ such that $addr_{\tau'} = addr_\tau$, then select the smallest such τ' and set $r_\tau^{sim} \leftarrow l_{2\tau'}^{sim}$. Otherwise, set r_τ^{sim} to a randomly chosen leaf node. More specifically, the simulator computes r_τ^{sim} as follows:
   (a) Initialize the set $A := \{(\tau, addr_\tau, l_{2\tau}^{sim})\}_{\tau \in [T]}$.
   (b) Compute the list A' using A such that all the tuples in A are sorted based on $(addr_\tau, \tau)$ i.e. the elements in A are first sorted based on $addr_\tau$ and all the tuples with the same $addr_\tau$ are sorted based on τ. Let $A'[i]$ denote the i^{th} tuple of A.
   (c) Initialize R to empty list of size T. For each $i \in [T]$, the $R[i]$ is computed as follows:
      – Set $(\tau, addr, l) := A'[i]$ and $(\tau', addr', l') := A'[i+1]$.
      – If $addr = addr'$, set $R[i] = (\tau, l')$; otherwise set $R[i] := (\tau, rand)$ where $rand$ is a randomly chosen leaf node.
   (d) Lastly, sort R based on the first element in the tuple i.e. the time step τ. Let the sorted list be $r_{eq} := \{r_1^{sim}, ..., r_T^{sim}\}$
5. Output r_{eq}.

[a] We are assuming that the RAM program P accesses at memory every time step; so the number of memory accesses m made by P is equal to T.

Fig. 2. Description of Simulator Sim_2, which outputs the randomness used to compute the memory access pattern of a ORAM.

algorithm Sim_2 in Figure 2. If an overflow occurs, Sim_2 outputs *overflow* instead of outputting the randomness r_{eq}. This does not violate adaptive security as overflow occurs with just negligible probability. The adaptive security hence follows from the existence of algorithms Sim_1 and Sim_2.

Cost Analysis of Sim_2. The cost of Sim_2 is $O(T \log T)$. This cost arises from the sort operations in steps $4(b)$ and $4(d)$ in Figure 2, which are used to compute two consecutive memory accesses that access the same memory address.

3 RAM-efficient Equivocal Encryption

In this section we define and construct *RAM-efficient Equivocal Encryption* (REE) which is similar to Functionally Equivocal Encryption from [9]. As motivated in the introduction, an REE scheme, as opposed to an FEE, provides a more efficient construction of an equivocal garbled RAM. Similar to an FEE, an REE is an equivocal encryption, meaning that the simulator can generate a dummy ciphertext (without knowing the plaintext $m \in \mathcal{M}$) and later equivocate it to some plaintext m', by providing randomness r_{Enc} and the key k consistent with plaintext m' and the dummy (or simulated) ciphertext. There are two main differences between an REE and an FEE. Firstly, FEE equivocates with respect to a function, an REE can equivocate with respect to a RAM program P. Secondly, FEE needs to equivocate based on the input of the function, whereas REE equivocates based on the input as well as the database corresponding to the RAM program. This implies that the key for an FEE comprises of just the garbled input while the key for an REE comprises of the garbled input along with a garbled database.

We begin with the description of the syntax of REE which comprises of the following algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{SimTrap}, \mathsf{SimEnc}, \mathsf{Equiv}, \mathsf{Adapt})$.

– **Key generation.** $\mathsf{REE.Gen}(1^\lambda, 1^n; r_{\mathsf{Gen}})$ takes as input security parameter λ, equivocation parameter n, and randomness r_{Gen} of size $\mathsf{poly}(\lambda, n)$. It sets the key $k := r_{\mathsf{Gen}}$ and outputs it.
  Note that the key size only depends on equivocation parameter and security parameter, but not on the plaintext size.
– **Encryption.** $\mathsf{REE.Enc}_k(\mathsf{params}, msg; r_{\mathsf{Enc}})$ takes as input params (which comprises of the description size of the RAM program s and memory size M) and plaintext msg. It outputs an encryption of m with respect to the params using randomness r_{Enc} and key k. Let $c = (\tilde{P}, \tilde{D})$ be the encryption of msg where $\tilde{P}$ and $\tilde{D}$ denote garbled RAM program and garbled memory respectively.
– **Decryption.** $\mathsf{REE.Dec}_k(c)$ decrypts ciphertext c using key k and outputs plaintext $m = P^D(x)$.
– **Ciphertext simulation.** Simulating a ciphertext comprises of two algorithms $(\mathsf{REE.SimTrap}, \mathsf{REE.SimEnc})$. Algorithm $\mathsf{REE.SimTrap}$ takes as input $(1^\lambda, 1^n; r_{\mathsf{td}})$ and outputs the trapdoor td. Algorithm $\mathsf{REE.SimEnc}$ on input $(\mathsf{params}, \mathsf{td}; r_{\mathsf{Sim}})$ outputs a ciphertext $c_{eq} = (\tilde{P}, \tilde{D})$ with respect to params (where params comprises of description of the RAM program P and memory size M).
– **Equivocation.** $\mathsf{REE.Equiv}(x, \mathsf{td})$ uses the equivocation trapdoor td to generate a single fake key k_{eq} so that each simulated ciphertext c_{eq} decrypts to $P^D(x)$ under k_{eq}. Note that the c_{eq} was generated with respect to some (P, D) and trapdoor td.
– **Randomness sampling.** $\mathsf{REE.Adapt}(P, D, \mathsf{td}, r_{\mathsf{Sim}}, x)$ generates randomness r_{eq}, such that $\mathsf{REE.Enc}_{k_{\mathsf{eq}}}(\mathsf{params}, P^D(x); r_{\mathsf{eq}}) = c_{eq}$ where $c_{eq} := \mathsf{REE.SimEnc}$ $(P, D, \mathsf{td}; r_{\mathsf{Sim}})$. In other words, the $\mathsf{REE.Adapt}$ algorithm comes up with

random coins such that the real garbled RAM program and garbled memory i.e. $(\tilde{P}, \tilde{D})$ look like they are simulated, which makes use of the obliviousness property of Yao's garbling scheme.

Security. The honestly generated encryptions and the simulated encryptions along with the messages, the random coins, and the key, are indistinguishable where the message is the output of the execution of $P^D(x)$, $x \in \{0,1\}^n$ is the input, P is a RAM program from $\{0,1\}^n \to \{0,1\}^\ell$ and D is the memory accessed by P. More formally, we need to show that for any PPT adversary $\mathcal{A}$ the following two distributions are indistinguishable:

$$\begin{aligned} D_0^n = \{&(P, D, x) \leftarrow \mathcal{A}(1^\lambda, 1^n); \mathsf{td} \leftarrow \mathsf{REE.SimTrap}(1^\lambda, 1^n; r_{\mathsf{td}}); \\ c_{\mathsf{eq}} &:= (\tilde{P}_{eq}, \tilde{D}_{eq}) \leftarrow \mathsf{REE.SimEnc}(\mathsf{params}, \mathsf{td}, r_{\mathsf{Sim}}); \\ &\qquad k_{\mathsf{eq}} \leftarrow \mathsf{REE.Equiv}(\mathsf{td}, x); r_{\mathsf{eq}} \leftarrow \mathsf{REE.Adapt}(P, D, \mathsf{td}, r_{\mathsf{Sim}}, x) : \\ &\qquad\quad (k_{\mathsf{eq}}, c_{\mathsf{eq}}, r_{\mathsf{eq}})\} \end{aligned}$$

$$\begin{aligned} D_1^n = \{&(P, D, x) \leftarrow \mathcal{A}(1^\lambda, 1^n); k \leftarrow \mathsf{REE.Gen}(1^\lambda, 1^n; r_{\mathsf{Gen}}); \\ &c := (\tilde{P}, \tilde{D}) \leftarrow \mathsf{REE.Enc}_k(\mathsf{params}, m; r_{\mathsf{Enc}}) : \\ &\qquad (k, c, r_{\mathsf{Enc}}))\} \end{aligned}$$

Overview of [17] We give an overview of the garbled RAM construction of [17] before presenting the REE. Refer to the full paper for a formal description.

Garbling the Data. Let $m = |D|$ and $d = \log(m/\kappa)$. The garbled data is in the form of a binary tree of keys of depth d. The *plain version* of this tree of keys comprises of data elements (of size κ) at the leaf nodes and random κ-bit values (which are used as PRF keys) at the non-leaf nodes. The *encrypted version* is computed from the *plain version* as follows: each non-root node $r \in \{0,1\}^\kappa$ is encrypted using its parent $s \in \{0,1\}^\kappa$ as the key: $F_s(\text{left/right}, k, r_k)$ (for leaf nodes, data element D_k is encrypted instead of r_k). The protocol GData$(1^\kappa, D)$ outputs the encrypted version of the tree of keys $\tilde{D}$ and the key corresponding to the root of the tree.

Garbling the Program. Let T be the running time of the program P. We need to garble each of the T CPU steps, which perform a read and write to memory. A CPU step at time step τ is denoted as follows: $C^P_{CPU}(\text{state}, z^{\text{read}}) = (\text{state}', L, z^{\text{write}})$ where L' is the memory location to be read at the next time step i.e. $\tau + 1$, z^{write} is the value to be written into the location L' in the next time step i.e. $\tau + 1$ and z^{read} is the value read from the memory location L, where L is the memory location output by the previous time step $\tau - 1$. Here, a simplifying assumption is that the read and write is made to the same memory location L. For further ease of exposition, the protocol is provided assuming Unprotected Memory Access (UMA).

Without loss of generality, we focus on reading an element from location L and then executing the i^{th} CPU step. Reading an element from the database is done by navigating through the tree of keys all the way to the leaf nodes where the data is located (as described earlier). To traverse from the root to the leaf node of the tree, a sequence of *navigation* circuits C^{nav} are used, one for each level. C^{nav} works as follows: it takes as input two sibling PRF keys and chooses one of them (either the left or right PRF key) based on the location L to be read. Then, C^{nav} uses this chosen key to decrypt and outputs the keys corresponding to its children. These two child PRF keys are used as inputs to the next C^{nav} in the sequence. After navigating through $(d-1)$ level of the tree using C^{nav} circuits, the last level is processed using C^{step} circuit, which does the following:

– Executes CPU computation for the current time step.
– Writes data element z^{write} to location L'.
– Kick starts execution of the next time step by outputting the two PRF keys corresponding to children of the root. This serves as the input to C^{nav} at the start of the next time step.

The outputs of the navigation and step circuits consist of $(write, translate, aux)$ which we explain below.

1. **aux**: comprises of (state, L) where state denotes the state of the computation and L is the memory location to be read. This value is the output of the CPU computation step.
2. **translate**: enables reading of data. The goal is to generate the input label for the next circuit corresponding to the value read from memory. Since the memory location to be read is generated during runtime, *translate* can only be determined during the runtime.
3. **write**: enables writing of data. If the key corresponding to any node needs to be modified/written to a new key, it affects two nodes in the garbled memory: (1) the current node re-encrypts the keys of its children under the new key and (2) the parent of this node is update to store the encryption of the new key.

To overcome the circularity issues of Lu-Ostrovsky's construction, the PRF keys are replaced with fresh ones each time they are used to read. More specifically, the keys are replaced whenever they are used to compute *translate*. The tree structure of the garbled database limits the number of PRF keys that need to be replaced for each CPU step to be polylogarithmic in the memory size. Finally, the protocol $\mathrm{GProg}(1^\kappa, 1^{\log m}, 1^t, P)$ outputs the set of garbled circuits for each CPU step and the set of input keys s^{in}.

Garbling the Input and Evaluation. Let s be the PRF key corresponding to the root of the garbled database. The protocol for input garbling $GInput(1^\kappa, x, s^{in}, s)$ outputs *translate* to enable reading the PRF keys corresponding to the children of root s. Also, it outputs the selection of the labels corresponding to x in s^{in}. The evaluation of each circuit is similar to evaluating any garbled circuit. In addition, we need to obtain the input labels of a circuit from the output labels of the previous circuit in the sequence, which is done using *translate*. This is because the outputs of a circuit are passed as inputs to another circuit.

3.1 Our Construction

We now provide a formal description of the algorithms for our REE. Let $(\mathsf{Gen}^{\mathsf{CPA}}, \mathsf{Enc}^{\mathsf{CPA}}, \mathsf{Dec}^{\mathsf{CPA}})$ be a private-key encryption scheme with the following two properties: (1) has pseudorandom ciphertexts and (2) decrypting a random string with the key outputs a pseudorandom plaintext[9].

Key Generation: $\mathsf{REE.Gen}$ on input $(1^\kappa, 1^n; r_{\mathsf{Gen}})$ computes key k as follows:

1. Sample $\overline{\mathsf{lab}}^{inp} = (\overline{\mathsf{lab}}^{aux}, \overline{\mathsf{lab}}^{read})$ where $|aux|$ independent labels $\overline{\mathsf{lab}}^{aux} = (\overline{\mathsf{lab}}_1, \ldots, \overline{\mathsf{lab}}_{|aux|})$ and $|read| = 2\kappa$ independent labels $\overline{\mathsf{lab}}^{read} = (\overline{\mathsf{lab}}_1, \ldots, \overline{\mathsf{lab}}_{2\kappa})$ using $\mathsf{Gen}^{\mathsf{CPA}}(1^\kappa)$.
2. Set $translate^{inp} \leftarrow \mathsf{GenTranslate}(0, 1, 0^d, \overline{\mathsf{lab}}^{read})$ ($\mathsf{GenTranslate}$ is described in Fig. 3).
3. Output $k = (translate^{inp}, \overline{\mathsf{lab}}^{aux})$

Encryption: Enc with key $\overline{\mathsf{lab}}^{inp} = (\overline{\mathsf{lab}}^{aux}, \overline{\mathsf{lab}}^{read})$ on input $(P, D, y; r_{\mathsf{Enc}})$, where $|y| = l$, generates ciphertext c as follows:

1. Create a file $\mathsf{BookKeeping}$ to keep track of the nodes in the memory that were traversed during the RAM computation.
2. **Generate garbled program** $\tilde{P}$: Let $L^\tau \in [m]$ denote the location of the memory access in time step τ.
   For each *step* $\tau = t - 1$ to 0:
      For each *level* $i = d - 1$ to 0:
      - Sample labels in $\overline{\mathsf{lab}}^{\tau,i} = (\overline{\mathsf{lab}}^{\tau,i,aux}, \overline{\mathsf{lab}}^{\tau,i,read})$ using $\mathsf{Gen}^{\mathsf{CPA}}(1^\kappa)$.

τ	i	*tag*	*output*
$\{t-1\}$	$\{d-1\}$	final	$write, aux = (y, L^{t-1})$
$\{t-1, ..., 0\}$	$\{d-1\}$	step	$write, translate, \overline{\mathsf{lab}}^{\tau+1,0,aux}$
$\{t-1, ..., 0\}$	$\{d-2, ..., 0\}$	nav	$write, translate, \overline{\mathsf{lab}}^{\tau,i+1,aux}$

Table 1. Inputs to GSim for different time steps τ and levels i.

      - $write \leftarrow \mathsf{GenWrite}(\tau, i, L^\tau)$

[9] We note that the PRF-based scheme $(r, F_k(r) \oplus m)$ satisfies both the properties.

Subprotocols for $\mathsf{REE.Enc}$

$\mathsf{GenTranslate}(\tau, i, L, (\mathsf{lab}^{\,left,\,k}, \mathsf{lab}^{\,right,\,k}))$

– Set $r^{left/right,\,k} \leftarrow \{0,1\}^\kappa$ and $rand^{left/right,\,k} \leftarrow \{0,1\}^\kappa$ for all $k \in [\kappa]$. Output

$$translate := \left\{ \begin{matrix} r^{left,\,k} \oplus \mathsf{lab}^{left,\,k} & r^{right,\,k} \oplus \mathsf{lab}^{right,\,k} \\ rand^{left,\,k} & rand^{right,\,k} \end{matrix} \right\}_{k\in[\kappa]}$$

– Let l be the i highest order bits of L. Then add the following entries to the file $\mathsf{BookKeeping}$ for all $k \in [\kappa]$:
- $\hat{r}^{i+1,\,2l,\,k}\{\tau\} = r^{left,\,k}$
- $\hat{r}^{i+1,\,2l+1,\,k}\{\tau\} = r^{right,\,k}$

$\mathsf{GenWrite}(\tau, i, L)$

– Let l be the i highest order bits of L. Look through $\mathsf{BookKeeping}$ to find entries $\hat{r}^{i+1,\,2l,\,k}\{p\}$ and $\hat{r}^{i+1,\,2l+1,\,k}\{p\}$ such that $p > \tau$ and pick smallest such p if it exists.
– If such a p exists, set $\alpha^{left,\,k} = \hat{r}^{i+1,\,2l,\,k}\{p\}$ and $\alpha^{right,\,k} = \hat{r}^{i+1,\,2l+1,\,k}\{p\}$; else set $\alpha^{left,\,k} \leftarrow \{0,1\}^\kappa$ and $\alpha^{right,\,k} \leftarrow \{0,1\}^\kappa$ for all $k \in [\kappa]$.
– Output $(L, (\alpha^{left,\,k}, \alpha^{right,\,k}))$.

Fig. 3. Subprotocols for $\mathsf{REE.Enc}$

- $translate \leftarrow \mathsf{GenTranslate}(j, k, L^j, \overline{\mathsf{lab}}^{\,j,k,read})$ where j and k are chosen as follows:If $i = d-1$, set $j = \tau+1$, $k = 0$; otherwise, set $j = \tau$, $k = i+1$. The descriptions of $\mathsf{GenWrite}$ and $\mathsf{GenTranslate}$ are given in Figure 3.
- If $\tau = 0, i = 0$, then $\tilde{C}^{\tau,i} \leftarrow \mathsf{GSim}(1^\kappa, C^{tag}, \overline{\mathsf{lab}}^{inp}, output)$; otherwise $\tilde{C}^{\tau,i} \leftarrow \mathsf{GSim}(1^\kappa, C^{tag}, \overline{\mathsf{lab}}^{\tau,i}, output)$ where C^{tag} and y are chosen based on τ and i as per table 1.

3. Set $\tilde{P} := \{C^{\tau,i}$ for $\tau \in [t],\ i \in [d]\}$.
4. **Generate garbled memory** $\tilde{P}$: For all $i \in [d+1] - \{0\}$, $j \in [2^i]$, $k \in [\kappa]$, find the smallest entry $\hat{r}^{i,j,k}\{\tau\}$ in $\mathsf{BookKeeping}$ such that $\tau > 0$.
   – If such an entry exists, set $\hat{r}^{i,j,k} := \hat{r}^{i,j,k}\{\tau\}$; Else sample a uniformly random $\hat{r}^{i,j,k} \leftarrow \{0,1\}^\kappa$.
   – Set $D^{j,k} := \{\hat{r}^{d,j,k}$ for $j \in [2^d],\ k \in [\kappa]\}$.
5. Set $\tilde{D} := \{\hat{r}^{i,j,k}$ for all $i \in [d] - \{0\}$, $j \in [2^i]$, $k \in [\kappa]$, $D^{j,k}$ for $j \in [2^d]$, $k \in [\kappa]\}$.
6. Output ciphertext $c = (\tilde{P}, \tilde{D})$.

Decryption. Given ciphertext $c = (\tilde{P}, \tilde{D})$ and key $k = (translate^{inp}, \overline{\mathsf{lab}}^{aux})$, evaluate the simulated garbled RAM program $\tilde{P}$ with the key k using $\mathsf{GRAM.Eval}^{\tilde{D}}(\tilde{P}, k)$ and output the result.

Simulating ciphertexts. Let $n = (|aux| + |read|)$. $\mathsf{REE.SimTrap}(1^\kappa, 1^n; r_{\mathsf{td}})$ samples $2n$ keys using $\widehat{\mathsf{lab}}^{j,0}$ and $\widehat{\mathsf{lab}}^{j,1}$ for $1 \leq j \leq n$ using $\mathsf{Gen}^{\mathsf{CPA}}(1^\kappa)$ and outputs $\mathsf{td} = (\widehat{\mathsf{lab}}^{aux}, \widehat{\mathsf{lab}}^{read}) = (\widehat{\mathsf{lab}}^{1,0}, \widehat{\mathsf{lab}}^{1,1}, \ldots, \widehat{\mathsf{lab}}^{n,0}, \widehat{\mathsf{lab}}^{n,1})$.
$\mathsf{REE.SimEnc}$ on input $(P, \mathsf{td}; r_{\mathsf{Sim}})$ computes c_{eq} as follows:
1. Set $(\tilde{P}, root) \leftarrow \mathsf{GRAM.Prog}(1^\kappa, 1^{\log m}, 1^t, P, td)$
2. Set $\tilde{D} \leftarrow \mathsf{GRAM.Data}(1^\kappa, \mathsf{D})$
3. Finally, output $c_{\mathsf{eq}} = (\tilde{P}, \tilde{D})$.

Equivocation. $\mathsf{REE.Equiv}$ on input (x, td) uses the trapdoor $\mathsf{td} = (\widehat{\mathsf{lab}}^{aux}, \widehat{\mathsf{lab}}^{read})$ to compute the key $k_{\mathsf{eq}} = \mathsf{GRAM.Inp}(1^\kappa, x, td, root)$.

Randomness sampling. $\mathsf{REE.Adapt}$ on input $(P, D, \mathsf{td}, r_{\mathsf{Sim}}, x)$ needs to generate a random string r_{eq} so that $\mathsf{REE.Enc}_{k_{\mathsf{eq}}}(\mathsf{params}, P^D(x); r_{\mathsf{eq}}) = c_{\mathsf{eq}} = (\tilde{P}, \tilde{D})$. To generate r_{eq}, it proceeds as follows:
1. Let $\tilde{P} = (\tilde{C}_1, \ldots, \tilde{C}_t)$ and $(\widehat{\mathsf{lab}}^1, \ldots, \widehat{\mathsf{lab}}^t)$ denote the input labels which are determined while generating the RAM program using GRAM.Prog. Also, let $(x_1, \ldots, x_t)$ denote the inputs corresponding to the circuits and are determined while evaluating $P^D(x)$.
2. $r_P = (oSamp(1^\kappa, k_1, x_1), \ldots, oSamp(1^\kappa, k_t, x_t))$
3. $r_D = \{\hat{r}^{i,j}\ \forall i \in [d], j \in [2^i]\}$
4. For each circuit $\tilde{C}^{\tau,i-1}$, let the $translate^{\tau,i-1}$ be:

$$\begin{Bmatrix} F_{r^{i-1,\lfloor j/2 \rfloor}}(left, k, 0) \oplus \mathsf{lab}^{left,k,0} & F_{r^{i-1,\lfloor j/2 \rfloor}}(right, k, 0) \oplus \mathsf{lab}^{right,k,0} \\ F_{r^{i-1,\lfloor j/2 \rfloor}}(left, k, 1) \oplus \mathsf{lab}^{left,k,1} & F_{r^{i-1,\lfloor j/2 \rfloor}}(right, k, 1) \oplus \mathsf{lab}^{right,k,1} \end{Bmatrix}_{k \in [\kappa]}$$

The $translate$ given above is used to read the labels for nodes $(r^{i,j,k}, r^{i,j+1,k})$ and is revealed as:

$$r^{\tau,i-1}_{translate} = \begin{Bmatrix} r^{left,k} \oplus \mathsf{lab}^{left,k} & r^{right,k} \oplus \mathsf{lab}^{right,k} \\ rand^{left,k} & rand^{right,k} \end{Bmatrix}_{k \in [\kappa]}$$

where $r^{left,k} = F_{r^{i-1,\lfloor j/2 \rfloor}}(left, k, r^{i,j,k})$ (corresponds to the active row) and $rand^{left,k} = F_{r^{i-1,\lfloor j/2 \rfloor}}(left, k, 1 \oplus r^{i,j,k}) \oplus \mathsf{lab}^{left,k,1}$. Similarly, $r^{right,k}$ and $rand^{right,k}$ are defined.

5. For each circuit $C^{\tau,i}$, let the $write$ be:

$$write^{\tau,i} = (F_{r^{i-1,\lfloor j/2 \rfloor}}(left, k, r^{i,j,k}), F_{r^{i-1,\lfloor j/2 \rfloor}}(right, k, r^{i,j+1,k}))$$

be revealed as

$$r^{\tau,i}_{write} = (r^{left,k}, r^{right,k})$$

6. Let $r_{translate} = \{r^{\tau,i}_{translate} \mid \forall \tau \in [t], i \in [d]\}$ and $r_{write} = \{r^{\tau,i}_{write} \mid \forall \tau \in [t], i \in [d]\}$
7. Output $r_{eq} = (r_P, r_D, r_{translate}, r_{write})$.

Theorem 5. *Assuming the encryption scheme is CPA-secure, REE comprising of* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{SimTrap}, \mathsf{SimEnc}, \mathsf{Equiv}, \mathsf{Adapt})$ *is a RAM-efficient Equivocal Encryption scheme with* $\tilde{O}(T + M + n)$ *decryption time complexity and* $\tilde{O}(T + M + n)$ *ciphertext size where* T*,* M *and* n *are the running time, memory size and input size of the RAM program* P *respectively.*

The proof of this theorem is similar to proof of security of the Garbled RAM from [17] and is given in the full version of our paper.

4 Equivocal Garbled RAM

In this section we discuss how to construct an equivocal garbled RAM by extending [9]'s framework to RAM programs. Roughly speaking, to equivocate RAM programs, we use REE instead of FEE to encrypt the rows of the garbled gates. The garbled RAM construction of [16] primarily involves "communicating" garbled circuits. So the problem of obtaining an equivocal garbled RAM can be reduced to adaptively garbling each of the sub-circuits. In this section, we focus on how to garble the sub-circuits using REE with equivocation. We first define an equivocal garbling scheme for RAM programs and then give an overview of the [16] construction.

Definition 1 (Equivocal garbling scheme for RAM programs). *We say that* GRAM *comprising of* $(Data, Prog, Inp, Eval)$ *is an equivocal garbling scheme for RAM programs, if the following properties hold:*

- **Correctness:** *For any RAM program* P*, Data* D *and input to the RAM program* x *we require the following to hold:*

  $$\mathsf{Pr}[r \leftarrow \{0,1\}^{|r|}; K \leftarrow \mathsf{Gen}(1^\lambda); \widetilde{D}, root \leftarrow \mathsf{GRAM.Data}(1^\kappa, D), \widetilde{P} \leftarrow \mathsf{GRAM.Prog}$$
  $$(1^\kappa, K, P, root; r); \{\widetilde{x}\} \leftarrow \mathsf{GRAM.Inp}(1^\kappa, K, x) :$$
  $$\mathsf{GRAM.Eval}(\widetilde{P}, \widetilde{D}, \widetilde{x}) = P^D(x)] > 1 - \mathsf{negl}(\lambda);$$

- **Security:** *There exists a pair of PPT algorithm* $(\mathsf{Sim}_1, \mathsf{Sim}_2)$*, such that any PPT adversary* A *wins the following game with at most negligible advantage:*
  1. *A gives a RAM program P, memory D and an input x to the challenger;*
  2. *The challenger flips a bit b.*
     If $b = 0$*:*
     - *It chooses random garbling key K and randomness r;*
     - *It sets* $(\widetilde{D}, root \leftarrow \mathsf{GRAM.Data}(1^\kappa, D), \widetilde{P} \leftarrow \mathsf{GRAM.Prog}(1^\kappa, K, P, root; r),$ $\{\widetilde{x}\} \leftarrow \mathsf{GRAM.Inp}(1^\kappa, K, x)$*;*
     - *It sends* $\widetilde{P}, \widetilde{D}, \widetilde{x}, K, r$ *to the adversary.*

     If $b = 1$*:*
     - *It sets* $y = P^D(x)$*;*
     - *It runs the simulator* $(\widetilde{P}, \widetilde{D}, \widetilde{x}, \mathsf{state}) \leftarrow \mathsf{Sim}_1(P, D, y)$
     - *It runs the simulator* $(K_{eq}, r_{eq}) \leftarrow \mathsf{Sim}_2(\mathsf{state}, x)$
     - *It sends* $\widetilde{P}, \widetilde{D}, \widetilde{x}, K_{eq}, r_{eq}$ *to the adversary.*
  3. *The adversary outputs a bit b'.*

  The adversary wins if $b = b'$*.*

Overview of [16] Garg et. al. [16] presented a black box approach to garble RAM programs. Roughly, their construction represents the entire RAM program (including the memory) as a set of circuits which are then garbled. Each circuit *communicate* with the *next* circuit by outputting the input labels corresponding to the *next* circuit. Looking ahead, the observation that all the garbled circuit just output labels corresponding to the input of other garbled circuits in [16] is crucial to the construction of our Equivocal RAM construction.

The mechanism for enabling memory access in [16] is to maintain the garbled data as a *tree* of garbled circuits, known as *memory circuits*, that can *communicate* with the neighboring garbled circuits. This *communication* happens when a garbled circuit outputs the appropriate input labels corresponding to the garbled circuit it intends to communicate with. In order to read/write, the garbled RAM program first passes the control to the root circuit of the garbled database. Depending on the location to be read/written, control is passed through a path of garbled circuits from the root to the leaf node that stores the data that needs to read or written. The leaf garbled circuit, which stores the data, passes control back to the RAM program along with labels corresponding to the data. Over the course of the RAM computation, the control may be passed multiple times to the root garbled circuit, once for each read/write. However, garbled circuits offer no security if they are used more than once i.e., they are evaluated over multiple inputs. So, the root garbled circuit must be refreshed to maintain security. This refreshing of garbled circuits needs to be done with care as it should maintain the property of each of the circuit to communicate with its *neighboring* circuit. This results in the following two main challenges: (i) the *used* garbled circuits must be replaced with fresh ones and (ii) the garbled circuit must *know* the input labels of the circuit it communicates with.

The first issue involves replacing the entire path of garbled circuits from the root to the leaf node with fresh garbled circuits for every memory access. A simple approach of replacing garbled circuits is as follows. If there were T memory accesses overall, then let each node of the garbled tree can have T garbled circuits such that the i^{th} garbled circuit in any node can communicate with the i^{th} garbled circuit at the left and right child nodes. This approach will work but comes at a prohibitively high cost, namely the garbled data size will be $O(TM)$. [16] observed that in this solution most of the garbled circuits are not used, especially at layers closer to the leaf nodes. All the T circuits at the root node are used up because the root is accessed for each of the T steps. But at the next layer there are $2T$ garbled circuit of which only T are used. If we assume memory accesses form a uniform distribution, then each of the left and right children in the first layer would use $T/2$ garbled circuits on average and each node in the second layer would use $T/4$ garbled circuits on average and so on. The reduction in the number of circuits in the subsequent number of layers needs to be done carefully taking into consideration that the number of circuits used can deviate from the expectation. By carefully bounding the number of garbled circuits at each node in every level one can ensure that the probability with which the circuits at these nodes will be over-consumed is negligible while still being efficient. [16] shows that the number of garbled circuits reduces from $O(MT)$ to $O(M)$.

The second issue is to allow a garbled circuits to communicate with "next" garbled circuit within this tree structure of garbled data. In slightly more detail, the garbled data is represented as a tree with each node comprising of a sequence of garbled circuits. A garbled circuit at any node can communicate with (i) its successor i.e. the garbled circuit that is next in sequence at that node, and (ii) its children (more specifically, a window of κ garbled circuits in each of the left and right child nodes). Enabling this communication between garbled circuits is one of the key aspects of the $\mathsf{GRAM.Data}$ algorithm. Suppose a garbled circuit C^A needs to communicate with another garbled circuit C^B, then C^A needs to output the input labels for C^B. C^A can either have the input labels of C^B passed as input to it or hard-coded within it. The key aspect of the $\mathsf{GRAM.Data}$ is to ensure that the garbled circuits have the appropriate labels to be able to communicate with other circuits. Now we look at the two main types of circuits present in the tree, one corresponding to the internal nodes and the other to the leaf nodes of the tree, which we briefly describe below.

Internal nodes. Each internal node of the tree is associated with a sequence of circuits, each of which are denoted by C^{node}. These circuits help in navigating control from the root to the leaf node which ultimately enables reading from or writing to memory. Any given circuit can pass control to (1) the next circuit at the same node, (2) one of the circuits located in its left child node or (3) a circuit located in its right child node. The circuits need to know the input labels of the circuits to which control is passed. So, each C^{node} circuit has input labels, corresponding to the three types of circuits mentioned above, hardcoded within it. This elaborate set of connections between the circuits is needed to refresh the circuits as and when they are used up.

Leaf nodes. The sub-circuits associated with the leaf nodes, say C^{leaf}, enable reading and writing of data into memory. Each leaf node comprises of a sequence of garbled circuits, similar to the internal nodes. The data is stored in these circuits by continually receiving it as input from its predecessor. When the data needs to be read, the C^{leaf} outputs the labels corresponding to data that are later fed into the CPU step circuit. In addition to outputting the input labels for CPU step circuit, it also passes on the data labels as input to the next C^{leaf} sub-circuit in the sequence. To write data into memory, the labels corresponding to the new data are passed as inputs to its successor.

Garbled Program. Garbling a RAM program P essentially involves garbling sub-circuits that carry out CPU computations at every time step. Each of these garbled step circuits outputs the labels for the new CPU state and the memory access information. The new CPU state is fed as input to the next step circuit. The memory access information comprises of labels of (1) the root circuit of the garbled data, (2) the location to read/write and (3) the data to be written.

The labels for the root circuit enable passing control from the root to the leaf nodes of the tree of circuits. The leaf nodes store the labels corresponding to the data to be read, which is passed as input to the next step circuit. Thus, the next step circuit receives the labels corresponding to the data as well as the CPU state, which are enough to proceed to the next time step.

Equivocating GRAM. At a high-level, we garble circuits output by the [16] construction using the equivocal garbling scheme of [9] where the underlying encryption scheme is instantiated with an REE (as opposed to an FEE). We provide the details of our construction and a proof sketch below.

4.1 Our Construction

We now present our equivocal garbled RAM construction.

Conventions. Consider a RAM program P with memory D and inputs x with running time T. Let $n = |x|$. We denote the output of the RAM program by $P^D(x)$. We use [16] construction to garble the RAM program and let the garbled versions of these programs be denoted by $\tilde{P}, \tilde{D}, \tilde{x}$.The garbled program and data comprise of three main sub-circuits: C^{step}, C^{node} and C^{leaf}. For the rest of this section, we show how to garble C which could potentially be any of the sub-circuits. Let κ be the security parameter. For any wire w in gate g, let k^0_w, k^1_w be the λ-bit labels associated with the wires where $\lambda = n\kappa + 1$ and bit_w be the actual bit assigned to w during $P^D(x)$. We consider some gate g with input wires α, β and output wire γ. We also assume without loss of generality that all gates are fan-in two gates.

Garbling Data and Program. $\mathsf{GRAM.Data}(1^\kappa, m)$ outputs the garbled data $(\tilde{\mathsf{D}}, root)$ and $\mathsf{GRAM.Prog}$ $(1^\kappa, 1^{\log m}, P, root)$ outputs $(\tilde{\mathsf{P}}, s^{in})$. At a high-level, we follow the garbled RAM construction of [16] to generate $\tilde{\mathsf{P}}$ and $\tilde{\mathsf{D}}$ with two key technical differences. First, we use an approach similar to equivocal garbling [8] to garble the sub-circuits instead of Yao's Garbling [30]. Secondly, the encryption scheme used to garble the gates is REE instead of FEE used [8] or regular CPA-secure encryption scheme used in Yao's construction.

Recall that the GRAM.Data and GRAM.Prog comprises of garbled circuits communicating with other garbled circuits. So we show how to garble the sub-circuits used in $\tilde{\mathsf{P}}$ and $\tilde{\mathsf{D}}$. Let C be some sub-circuit with n'-bit input string x_{sub}. We denote by $m = n + \mathsf{gates}(C)$ the total number of wires in C where gates with fan-out more than 1 are counted only once.

We first generate two labels (k^0_w, k^1_w) for every wire w in C using $\mathsf{REE}.Gen$ and then the garbler generates the following 4 pairs of ciphertexts for each gate g:

$$c^{00}_{g,\mathsf{left}} = \mathsf{REE.Enc}_{k^0_\alpha}(s^{00}_{g,\mathsf{left}}), \quad c^{00}_{g,\mathsf{right}} = \mathsf{REE.Enc}_{k^0_\beta}(s^{00}_{g,\mathsf{right}}),$$

$$c^{01}_{g,\mathsf{left}} = \mathsf{REE.Enc}_{k^0_\alpha}(s^{01}_{g,\mathsf{left}}), \quad c^{01}_{g,\mathsf{right}} = \mathsf{REE.Enc}_{k^1_\beta}(s^{01}_{g,\mathsf{right}}),$$

$$c^{10}_{g,\mathsf{left}} = \mathsf{REE.Enc}_{k^1_\alpha}(s^{10}_{g,\mathsf{left}}), \;\; c^{10}_{g,\mathsf{right}} = \mathsf{REE.Enc}_{k^0_\beta}(s^{10}_{g,\mathsf{right}}),$$

$$c^{11}_{g,\mathsf{left}} = \mathsf{REE.Enc}_{k^1_\alpha}(s^{00}_{g,\mathsf{left}}), \;\; c^{11}_{g,\mathsf{right}} = \mathsf{REE.Enc}_{k^1_\beta}(s^{11}_{g,\mathsf{right}}),$$

Garbling Inputs. $\mathsf{GRAM.Inp}(1^\kappa, x, s^{in}, s)$ outputs $\tilde{\mathsf{x}}$ which are the labels corresponding to the input x.

Garbled Evaluation. $\mathsf{GRAM.Eval}(\tilde{\mathsf{P}}, \tilde{\mathsf{D}}, \tilde{\mathsf{x}})$ evaluated the garbled program and outputs $\tilde{\mathsf{P}}^{\tilde{\mathsf{D}}}(\tilde{\mathsf{x}})$. This function is similar to [16] except that REE.*Dec* is used to decrypt the garbled gates.

Simulation. The simulation has two main parts: (i) simulating the garbled program $\tilde{\mathsf{P}}$, garbled data $\tilde{\mathsf{D}}$ and garbled inputs $\tilde{\mathsf{x}}$ such that $\tilde{P}^{\tilde{D}}(x) = P^D(x)$ and (i) simulating the internal randomness consistent with the revealed inputs.

Simulation of Garbled Program, Garbled Data and Garbled Input. The first step of the simulator is to run $ORAM.Sim_1(T, m)$ and obtain the sequence of memory accesses $\overrightarrow{M}$. Similar to the real garbling, it's enough to focus on how to simulate each of the garbled sub-circuits, say C, of $\tilde{\mathsf{P}}$ and $\tilde{\mathsf{D}}$. For each wire w in C, the simulator chooses $n\kappa$-bit REE keys, random bit Λ_w and an REE trapdoor $\mathsf{td}_w \leftarrow \mathsf{REE.SimTrap}(1^\kappa, n)$. The simulated garbled gate is computed using REE.*Enc* and REE.*SimEnc* as shown in table 3. Note that each RAM program $P_{C^{\mathsf{type}},g}$ used in REE.SimEnc has $P, C^{\mathsf{type}}, g, k_\gamma, \mathsf{td}_\gamma, \Lambda_\gamma, \overrightarrow{M}, \tau, m$ hardcoded within its description, where γ is an output wire of g.

Row number	Left ciphertext REE	Right ciphertext REE
$(\Lambda_\alpha, \Lambda_\beta)$	$\mathsf{Enc}_{k_\alpha}(s^{\Lambda_\alpha,\Lambda_\beta}_{g,\mathsf{left}})$	$\mathsf{Enc}_{k_\beta}(s^{\Lambda_\alpha,\Lambda_\beta}_{g,\mathsf{right}})$
$(\Lambda_\alpha, 1 \oplus \Lambda_\beta)$	$\mathsf{Enc}_{k_\alpha}(s^{\Lambda_\alpha,1\oplus\Lambda_\beta}_{g,\mathsf{left}})$	$\mathsf{SimEnc}(P^{D_{in}}_{in}[\mathsf{prms}, s^{\Lambda_\alpha,1\oplus\Lambda_\beta}_{g,\mathsf{left}}])$
$(1 \oplus \Lambda_\alpha, \Lambda_\beta)$	$\mathsf{SimEnc}(P^{D_{in}}_{in}[\mathsf{prms}, s^{1\oplus\Lambda_\alpha,\Lambda_\beta}_{g,\mathsf{right}}])$	$\mathsf{Enc}_{k_\beta}(s^{1\oplus\Lambda_\alpha,\Lambda_\beta}_{g,\mathsf{right}})$
$(1 \oplus \Lambda_\alpha, 1 \oplus \Lambda_\beta)$	$\mathsf{SimEnc}(\mathsf{Const}[\mathsf{prms}, s^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}_{g,\mathsf{left}}])$	$\mathsf{SimEnc}(P^{D_{in}}_{in}[\mathsf{prms}, s^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}_{g,\mathsf{left}}])$

Table 2. Garbled gate g generated by the simulator.

The simulator orders 4 rows of each garbled gate as per $(\Lambda_\alpha, \Lambda_\beta)$ and outputs the 8 ciphertexts. Lastly, the garbled input comprising of the labels $(k_1, \ldots, k_n)$ corresponding to the input x (which are the active labels) are output by the simulator.

Simulation of the internal state of the garbler. To simulate the internal state of the garbled, we need to present the two main components: (i) the inactive keys $\widehat{k}_w$ for each wire w and (ii) randomness used to generate $\tilde{\mathsf{P}}, \tilde{\mathsf{D}}$ and $\tilde{\mathsf{x}}$ and is consistent with the input. We start by focusing on obtaining the internal sate for the garbled sub-circuits used with the $\tilde{\mathsf{P}}$. This approach can be applied to all the sub-circuits in order to obtain the internal state of the garbler.

Given the input x and data D, the modified input for the program P_{in} is $x||D$. The inactive keys are chosen as $\widehat{k}_w$ for each wire w. It generates these keys by running $\widehat{k}_w \leftarrow \mathsf{REE.Equiv}(\mathsf{td}_w; \hat{x})$ for each wire w. Now the garbling of P_{in} looks like the real garbling with keys $k_\alpha, \widehat{k}_\alpha, k_\beta, \widehat{k}_\beta$ where k_α, k_β are active for the computation $P_{in}^{D_{in}}(\hat{x})$.

The second component of the internal state is to present the randomness used in the garbling which is essentially the randomness used to encrypt the ciphertexts in each of the garbled gates. The simulator presents all the randomness used for encryption, a pair of keys per gate as internal state of the garbler and 8 secret shares ciphertexts per garbled gate.

Theorem 6. *Assuming the existence of RAM-efficient equivocal encryption,* GRAM *comprising of* $(Data, Prog, Inp, Eval)$ *is a Equivocal Garbled RAM scheme with a garbled database size of* $\tilde{O}((M+n+T)\cdot M)$, *garbled input size of* $\tilde{O}((M+n)\cdot n)$, *garbled program size and evaluation time of* $\tilde{O}((M+n+T)\cdot T)$, *where* T, M *and* n *are the running time, memory size and input size of the RAM program* P *respectively.*

Proof Sketch. The correctness of our Equivocal garbled RAM follows from the correctness of the Garbled RAM construction of [16] along with the correctness of the underlying REE. By induction, at each step, the evaluator gets the correct key $k_\gamma^{\mathsf{bit}_\gamma}$ and the correct pointer Λ_γ for the next gate's row.

Description of Hybrids. We define a sequence of hybrids H_0, H_0^{oram}, H_1, ..., H_t, H_{t+1}. The first hybrid H_0 corresponds to the real execution and the hybrid H_{t+1} to simulation. Further, we define m sub-hybrids between each H_i and H_{i+1} where $i \in \{1, ..., t-1\}$: $H_{i,1}^{subcirc}$, ..., $H_{i,m}^{subcirc}$ where $H_{i,m}^{subcirc}$ switches the key $k_{m-i}^{\mathsf{bit}_{m-i}}$ from real to simulated. Here the wires are sorted according to the topological order of the circuit, i.e. that output wires of each gate have larger index than both input wires of that gate (note that our notation $1, \ldots, n$ for input wires and m for an output wire is consistent with topological order). The descriptions of these hybrids are provided below.

Hybrid H_0^{oram}. In this hybrid we change how the permutation of ciphertexts is generated, without changing the distribution of the hybrid. Instead of generating the memory access sequence by evaluating the program $P^D(x)$, we generate it using the ORAM simulator Sim_1 . The garbled $\tilde{P}, \tilde{D}, \tilde{x}$ are generated as follows:

1. Generate the memory access $\overrightarrow{M} \leftarrow ORAM.Sim_1(T, m)$.
2. Compute the randomness $r_{oram} \leftarrow ORAM.Sim_2(\overrightarrow{M})$
3. Compute $P_{oram} \leftarrow ORAM.Prog(P, r_{oram})$ and garbled data $D_{oram} \leftarrow ORAM.Data(D)$

Description of Program $P_{in}[\mathsf{prms}, \mathsf{mask}]$ and Memory D_{in}

Constants: $P, k_\gamma, \mathsf{td}_\gamma, \Lambda_\gamma, \overrightarrow{M}, \tau, m, \mathsf{prms} = \{\tau, C^{\mathsf{type}}, g, b_\alpha \oplus \Lambda_\alpha, b_\beta \oplus \Lambda_\beta\}$
Input: x_{in}
Description of D_{in}. Initialize D_{in} to be an empty database of size $|D|$.
Description of P_{in}. The description of P_{in} follows.

1. Modify the program P to first write the input x into memory D_{in} and then proceed with the logic of P.
2. Run $ORAM.Sim_2[\overrightarrow{M}, P](D, x_{in})$ and write the output r_{eq} onto the random tape of the RAM program.
3. Run the compiler $ORAM.Prog$ with input P and randomness r_{eq}. Let the resulting program be referred to as P'.
4. For every time step $\tau \in T$, the bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires α, β of gate g are inferred depending on the type of circuit this gate belongs to:
   Cases C^{leaf} and C^{step}: Evaluate the program P' using memory D_{in} with some input to compute the bit assignments corresponding to the inputs of C^{leaf} and C^{step} circuits. Then the bit assignments of the gate g within these circuits can be computed using the inputs.
   Case C^{node}: The bit assignments of all the inputs to C^{node} have already been computed and hence the bits corresponding to gate g can be computed from the inputs without having to evaluate the program P' using memory D_{in} with some input.
5. Generate $\widehat{k}_\gamma \leftarrow \mathsf{REE.Equiv}(\mathsf{td}_\gamma, x)$. If $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) = g(b_\alpha \oplus \mathsf{bit}_\alpha, b_\beta \oplus \mathsf{bit}_\beta)$ then output $k_\gamma \oplus \mathsf{mask}$. Else output $\widehat{k}_\gamma \oplus \mathsf{mask}$.

Description of Program $\mathsf{Const}[\mathsf{const}]$

The program is padded to the size of programs P_{in} with memory D_{in} and is the RAM program that outputs the constant const.

Fig. 4. RAM programs and memory used in REE simulation

4. Lastly, compute $\tilde{P} \leftarrow \mathsf{GRAM.Prog}(P_{oram})$, $\tilde{D} \leftarrow \mathsf{GRAM.Data}(D_{oram})$ and $\tilde{x} \leftarrow \mathsf{GRAM.Inp}(x)$

The hybrid H_0^{oram} has a distribution similar to he real execution.

Hybrid H_i for $i = 1, ..., t$. Let the sequence of circuits evaluated during the execution of $GRAM.Eval(\tilde{P}, \tilde{D}, \tilde{x})$ be $C_1, ..., C_t$. In hybrid H_i, the first i circuits are simulated and the rest are generated as per the real execution. More specifically, this hybrid is computed as follows: simulate the first i circuits in the reverse order i.e. $C_i, ..., C_1$. This sequence of hybrids are identical to the ones considered in [16]. The main observation from [16] is that in Hybrid H_i, we can replace the real garbling of the i^{th} circuit with a simulated one as the input labels of the i^{th} circuit have been decoupled from the rest of the outputs. We will pursue the same approach with the exception that

our real and simulated garbling schemes are according to the equivocal garbling procedure [9] with REE encryption.
We consider a sequence of sub-hybrids following [9] between the hybrids H_i and H_{i+1} for $i = 1, ..., t-1$. The indistinguishability follows essentially as in [9], we present the hybrids explicitly rather than defining an equivocal garbling scheme instantiated with the REE.

Sub-Hybrid $H_{i,j}^{subcirc}$ for $j = 1, ..., m$. In this hybrid, the aim is to replace the wires in real garbled circuit C_{i+1} with simulated garbled circuits identical to the hybrids defined in [8] with the difference that REE is used instead of FEE. The real keys k_j corresponding to the j^{th} wire in circuit C_{i+1} are replaced with simulated keys. The ciphertexts corresponding to the simulated labels $\widehat{K}_j$ are also re simulating the ciphertext generated using these simulated keys. In this hybrid we convert the real labels $k_j^{1 \oplus \mathsf{bit}_j}$ to simulated labels. We essentially replace all the invocations to FEE functions with that of REE function in the hybrids of [8]. The indistinguishability of sub-hybrids $H_{i,j}^{subcirc}$ and $H_{i,j+1}^{subcirc}$ reduces to the security of the underlying REE scheme which has already been shown to be secure.

Hybrid H_{t+1}. In the previous hybrids H_t only the circuits executed during $\mathsf{GRAM.Eval}(\tilde{P}, \tilde{D}, \tilde{x})$ have been simulated so far. In this hybrid, those garbled circuits that haven't been evaluated will be considered. Note that some of these garbled circuits that weren't evaluated may contain labels to a (proper) subset of the input wires (i.e. partial inputs). These circuits can be topological ordered such that the outputs any circuit only goes as input to the subsequent circuits in the ordering and simulated in the reverse topological ordering.

4.2 Putting it together

We can essentially use an equivocal garbled RAM scheme in the standard Yao protocol assuming the existence of an adaptively secure oblivious-transfer protocol. The only difference from the standard construction is that we need to rely on the OT-functionality as a communication channel to transmit the garbled circuit and garbled inputs (of the the garbler). Another way of achieving this is to additionally assume a non-committing encryption (NCE) scheme and transmitting the data using NCE. We obtain the following theorem.

Theorem 7. *Let f be a two-party functionality expressed via a RAM program π. Assume the existence of a bit-decomposable equivocal garbled RAM scheme and the existence of a 2-round oblivious-transfer protocol secure against passive corruption by an adaptive adversary. Then there exists a 2-round 2-party protocol secure against passive corruption by an adaptive adversary where the communication complexity is $\tilde{O}(T^2 + n)$ where n is the sum of the input size of the two parties to f and T is the upper bound of the running time of π. Here $\tilde{O}(\cdot)$ ignores $\mathsf{poly}(\log T, \log n, \kappa)$ factors where κ is the (computational) security parameter.*

5 Adaptive Zero-Knowledge for RAM

In this section, we describe a simple construction of a UC zero-knowledge proof system for relations expressed as RAM computation. We build it from UC commitments and garbled RAM with a certain property which we call *splitability* We note that our construction also naturally works for circuits. We show that the GRAM construction of [17] is splitable in the full version.

Our proof system is RAM-efficient, meaning that its computation and communication complexity is only proportional to $\widetilde{O}(T)$. We then use the standard transformations [21,7] to compile any protocol from active to passive security. In particular, our protocol from theorem 7 results in an active protocol with computation and communication complexity $\widetilde{O}(T^2)$.

Theorem 8 (Adaptive ZK for RAM). *Assume the existence of a UC-secure commitment scheme secure against adaptive adversary, and the existence of static splitable garbled RAM. Then there exists a zero-knowledge proof of knowledge proof system secure against active corruption by an adaptive adversary where the communication complexity is $\widetilde{O}(T^2 + n)$ where n is the sum of the length of the statement and the witness, and T is the upper bound of the running time of the relation R_x. Here $\widetilde{O}(\cdot)$ ignores $\mathsf{poly}(\log T, \log n, \kappa)$ factors where κ is the (computational) security parameter.*

Theorem 9 (Adaptive, active 2PC for RAM). *Let f be a two-party functionality expressed via a RAM program π. Assume the existence of a bit-decomposable equivocal garbled RAM scheme, the existence of a UC-secure 2-round oblivious-transfer protocol secure against active corruption by an adaptive adversary, the existence of adaptive UC-secure commitment scheme, and the existence of static splitable garbled RAM. Then there exists a 2-party protocol secure against active corruption by an adaptive adversary where the communication complexity is $\widetilde{O}(T^2 + n)$ where n is the sum of the input size of the two parties to f and T is the upper bound of the running time of π. Here $\widetilde{O}(\cdot)$ ignores $\mathsf{poly}(\log T, \log n, \kappa)$ factors where κ is the (computational) security parameter.*

5.1 Splitable Garbling

The property of splitability can be defined both for garbled circuits and garbled RAM. For simplicity, we first informally describe it for circuits, but later give a formal definition for RAM programs (since this is what we use in the protocol); the differences are only syntactic.

Let C, x be a circuit and its input. Intuitively, splitability says that the "garbling information" G - i.e. garbled circuit and all labels - can be split into active part G^a and inactive part G^i, with the property that G^a is enough to learn the output $C(x)$. Further, we require that the garbling can be generated in different order: that is, one can either generate the full garbling information G and later, given x, compute G^a; or, one can generate G^a, without knowing x (only $C(x)$), and later complete it to full G, once w is given. We note that we require that the latter method generates the correct distribution of G and G^a (not merely an indistinguishable one). Next, we require verifiability: given G, it should be possible to determine which circuit (or, RAM program) it evaluates. Finally, we require input extractability: given G and G^a corresponding to input x, it should be possible to extract x in polynomial time.

For ease of exposition of our protocol, we also require that $G = G^a \cup G^i$ can be further split into $\{G^a{}_j\}, \{G^i{}_j\}$, such that, when they are shuffled, they jointly don't leak any information about the computation of C on x.

Syntax of splitable garbled RAM. For simplicity we assume that the memory D of the computation is empty (but there is still an input x). This will be sufficient for our ZK protocol.

– $G \leftarrow GRAM.Full(P; r_{\mathsf{Gen}})$ computes the full garbling information G;
– $\{G^a, I\} \leftarrow GRAM.Project(G, x)$ computes an active part G^a for computation $P(x)$, and a set of indices I such that G^a is a subset of G corresponding to positions I, i.e. $G^a = G_I$.
– $\{G^a, state, I\} \leftarrow GRAM.Active(P, y)$ computes an active part G^a for program P and its output y, together with positions I.
– $\{G, r_{\mathsf{Gen}}\} \leftarrow GRAM.Complete(G^a, x, state)$ computes the full garbling information G and proper generation randomness r_{Gen}, such that G^a is consistent with x.
– $y \leftarrow GRAM.Eval(G^a)$ computes the output y (supposedly equal to $P(x)$);
– $\{acc, rej\} \leftarrow GRAM.Verify(G, P)$ verifies whether G is a garbled RAM for the program P with empty memory;
– $x \leftarrow GRAM.Extract(G, G^a)$ outputs x such that $P(x) = GRAM.Eval(G^a)$.
– $GRAM.Perm(G)$ outputs a permuted description of G, such that revealing the location of G^a doesn't leak any information beyond $P(x)$.

Now we list the required properties:

– **Correctness**: For all P and x,

$$Pr[y \neq P(x) : r_{\mathsf{Gen}} \leftarrow \{0,1\}^{|r_{\mathsf{Gen}}|}, G \leftarrow GRAM.Full(P; r_{\mathsf{Gen}}),$$
$$G^a \leftarrow GRAM.Project(G, x), y \leftarrow GRAM.Eval(G^a)] \leq negl(\kappa)$$

– **Alternative generation:** For any x, P, and $y = P(x)$, the following distributions are the same:

$$\{(G, G^a, r_{\mathsf{Gen}}, I) : r_{\mathsf{Gen}} \leftarrow \{0,1\}^{|r_{\mathsf{Gen}}|}, G \leftarrow GRAM.Full(P; r_{\mathsf{Gen}}),$$
$$\{G^a, I\} \leftarrow GRAM.Project(G, x)\} \; and$$

$$\{(G, G^a, r_{\mathsf{Gen}}, I) : \{G^a, I, state\} \leftarrow GRAM.Active(P, y),$$
$$\{G, r_{\mathsf{Gen}}\} \leftarrow GRAM.Complete(G^a, x, state)\}.$$

– **Verifiability and Extractability:** For any P, any x, any string G and any substring $G^a = G_I$ (for some set I), if $GRAM.Eval(G^a) = y \neq \perp$, $GRAM.Verify(G, P) = acc$, and $x' \leftarrow GRAM.Extract(G, G^a)$, then $P(x') = y$. Naturally, we require that $GRAM.Verify(G, P) = acc$ for honestly generated G.

The adaptive UC zero-knowledge protocol

Prover's input: statement x and the corresponding relation R_x; witness w.
Verifier's input: statement x and the corresponding relation R_x.
The protocol:

1. The prover chooses random $r_{\mathsf{Gen}} \leftarrow \{0,1\}^{|r_{\mathsf{Gen}}|}$ and computes the splitable garbling of the relation R_x: $G = GRAM.Full(R_x; r_{\mathsf{Gen}})$. Then it computes $G^a, I \leftarrow GRAM.Project(G, w)$ and sets $G^i = G \setminus G^a$.
   The prover sends the verifier N UC commitments, where the committed value in commitment j is $Perm(G)_j$.
2. The verifier sends a random bit b to the prover.
3. If $b = 0$, the prover sends G together with decommitment information for all commitments to the verifier. If $b = 1$, the prover sends G^a together with decommitment information for indices $j \in I$ (which correspond to committed G^a) to the verifier.
4. If $b = 0$, the verifier accepts iff $GRAM.Verify(G, R_x)$ accepts and all decommitments verify. If $b = 1$, the verifier accepts iff $GRAM.Eval(G^a) = 1$ and the decommitments verify.

The simulation:

1. The simulator simulates N executions of the UC commitment.
2. If $b = 0$, the simulator generates $G = GRAM.Full(R_x; r_{\mathsf{Gen}})$, computes $Perm(G)$, and equivocates each commitment j, $j = 1, \ldots, N$, to G_j. It sets G and all the decommitments to be the simulated message of the prover. If $b = 1$, the simulator generates $G^a, I \leftarrow GRAM.Active(R_x, 1)$, equivocates commitments $j \in I$ to ${G^a}_j$ for each j, and sets G^a and the decommitment information to be the simulated message of the prover.
3. Upon corruption of the prover, the simulator learns w such that $R_x(w) = 1$. If $b = 0$, the simulator runs $G^a, I \leftarrow GRAM.Project(G, w)$ and sets $G^i = G \setminus G^a$. If $b = 1$, the simulator runs $\{G, r_{\mathsf{Gen}}\} \leftarrow GRAM.Complete(G^a, w, state)$ It sets $r_{\mathsf{Gen}}, G^a, G, I$, and the simulated coins of the commitment schemes to be the simulated state of the prover.

Fig. 5. The adaptive UC zero-knowledge protocol

5.2 Our Adaptive UC ZK Protocol

Our protocol is described on Figure 5. Completeness follows from correctness of the garbled RAM and correctness of UC commitment. We now explain why proof of knowledge and zero-knowledge properties hold.

Proof sketch. Intuitively, proof of knowledge (and soundness against unbounded provers) follows from extractability of splitable garbling and binding property of commitments. Let P be a prover which causes an honest verifier to accept with non-negligible probability. Consider an extractor which runs P till the end with verifier message 0, then rewinds P and runs it with verifier message 1. As a result, the extractor obtains G, G^a, and the decommitment information for both values, such that all decommitments verify, $GRAM.Verify(G, R_x) = acc$, and $GRAM.Eval(G^a) = 1$.

The extractor computes $w' = GRAM.Extract(G, G^a)$. By the binding property of commitments, it follows that $G^a = G_I$ for some set I. It then follows from extractability and from the fact that $GRAM.Verify(G, P) = acc$, $GRAM.Eval(G^a) = 1$, $R_x(w') = 1$.

To prove adaptive security, we need to simulate the protocol for an adversary which can adaptively corrupt potentially both parties (simulator described in Figure 5). Since the verifier is public-coin, it is enough to simulate the case where the adversary first lets the protocol finish and then corrupts the prover. Note that simulatability of this case immediately implies zero-knowledge, since the simulator doesn't have access to the witness when simulating the transcript. We argue that the simulation is computationally indistinguishable from the real execution.

- If $b = 0$, the simulator generates G and G^a honestly. The only difference between simulation and the real execution is that commitments are simulated and later equivocated to G. Indistinguishability follows immediately from the hiding property of commitments.
- If $b = 1$, the simulator uses alternative generation to generate G^a and later complete it to G. Recall that G, G^a, generated in this way, are required to be distributed correctly. Thus, the only difference between simulation and the real execution is that commitments are simulated and later equivocated to G. Indistinguishability follows from the hiding property of commitments.

6 Acknowledgments

Work of Rafail Ostrovsky was supported by NSF award #2001096, US-Israel BSF grant 2015782, a Google Faculty Award, a JP Morgan Faculty Award, an IBM Faculty Research Award, a Xerox Faculty Research Award, an OKAWA Foundation Research Award, a B. John Garrick Foundation Award, a Teradata Research Award, a Lockheed-Martin Research Award, and the Sunday Group.

References

1. Bellare, M., Hoang, V.T., Rogaway, P.: Adaptively secure garbling with applications to one-time programs and secure outsourcing. In: Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. pp. 134–153 (2012)
2. Benhamouda, F., Lin, H., Polychroniadou, A., Venkitasubramaniam, M.: Two-round adaptively secure multiparty computation from standard assumptions. In: Beimel, A., Dziembowski, S. (eds.) Theory of Cryptography - 16th International Conference, TCC 2018, Panaji, India, November 11-14, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11239, pp. 175–205. Springer (2018)

3. Bitansky, N., Canetti, R., Halevi, S.: Leakage-tolerant interactive protocols. In: Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings. pp. 266–284 (2012)
4. Bitansky, N., Dachman-Soled, D., Lin, H.: Leakage-tolerant computation with input-independent preprocessing. In: Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II. pp. 146–163 (2014)
5. Canetti, R., Feige, U., Goldreich, O., Naor, M.: Adaptively secure multi-party computation. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996. pp. 639–648 (1996)
6. Canetti, R., Goldwasser, S., Poburinnaya, O.: Adaptively secure two-party computation from indistinguishability obfuscation. In: Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II. pp. 557–585 (2015)
7. Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally composable two-party and multi-party secure computation. In: STOC. pp. 494–503 (2002)
8. Canetti, R., Poburinnaya, O., Venkitasubramaniam, M.: Better two-round adaptive multiparty computation. IACR Cryptology ePrint Archive **2016**, 614 (2016), http://eprint.iacr.org/2016/614
9. Canetti, R., Poburinnaya, O., Venkitasubramaniam, M.: Equivocating yao: constant-round adaptively secure multiparty computation in the plain model. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017. pp. 497–509 (2017)
10. Chung, K.M., Pass, R.: A simple oram. Tech. rep., CORNELL UNIV ITHACA NY (2013)
11. Cohen, R., Peikert, C.: On adaptively secure multiparty computation with a short CRS. In: Security and Cryptography for Networks - 10th International Conference, SCN 2016, Amalfi, Italy, August 31 - September 2, 2016, Proceedings. pp. 129–146 (2016)
12. Cohen, R., shelat, A., Wichs, D.: Adaptively secure MPC with sublinear communication complexity. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11693, pp. 30–60. Springer (2019)
13. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. Journal of Computer and System Sciences **7**(4), 354–375 (1973)
14. Dachman-Soled, D., Katz, J., Rao, V.: Adaptively secure, universally composable, multi-party computation in constant rounds. In: Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II. pp. 557–585 (2015)
15. Garg, S., Gupta, D., Miao, P., Pandey, O.: Secure multiparty RAM computation in constant rounds. In: Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I. pp. 491–520 (2016)
16. Garg, S., Lu, S., Ostrovsky, R.: Black-box garbled ram. Cryptology ePrint Archive, Report 2015/307 (2015), https://eprint.iacr.org/2015/307
17. Garg, S., Lu, S., Ostrovsky, R., Scafuro, A.: Garbled ram from one-way functions. Cryptology ePrint Archive, Report 2014/941 (2014), https://eprint.iacr.org/2014/941

18. Garg, S., Polychroniadou, A.: Two-round adaptively secure MPC from indistinguishability obfuscation. In: Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II. pp. 557–585 (2015)
19. Garg, S., Sahai, A.: Adaptively secure multi-party computation with dishonest majority. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7417, pp. 105–123. Springer (2012)
20. Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., Wichs, D.: Garbled RAM revisited. In: Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings. pp. 405–422 (2014)
21. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC. pp. 218–229 (1987)
22. Goyal, V.: Constant round non-malleable protocols using one way functions. In: STOC (2011)
23. Hazay, C., Venkitasubramaniam, M.: On the power of secure two-party computation. To Appear in CRYPTO 2016 **2016**, 74 (2016)
24. Hazay, C., Yanai, A.: Constant-round maliciously secure two-party computation in the RAM model. In: Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I. pp. 521–553 (2016)
25. Ishai, Y., Kumarasubramanian, A., Orlandi, C., Sahai, A.: On invertible sampling and adaptive security. In: Abe, M. (ed.) Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6477, pp. 466–482. Springer (2010). https://doi.org/10.1007/978-3-642-17373-8_27, https://doi.org/10.1007/978-3-642-17373-8_27
26. Lin, H., Pass, R.: Constant-round non-malleable commitments from any one-way function. In: STOC (2011)
27. Lindell, Y., Zarosim, H.: Adaptive zero-knowledge proofs and adaptively secure oblivious transfer. J. Cryptology **24**(4), 761–799 (2011)
28. Lu, S., Ostrovsky, R.: How to garble RAM programs. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7881, pp. 719–734. Springer (2013). https://doi.org/10.1007/978-3-642-38348-9_42, https://doi.org/10.1007/978-3-642-38348-9_42
29. Pippenger, N., Fischer, M.J.: Relations among complexity measures. Journal of the ACM (JACM) **26**(2), 361–381 (1979)
30. Yao, A.C.: Protocols for secure computations (extended abstract). In: FOCS. pp. 160–164 (1982)
31. Yao, A.C.: How to generate and exchange secrets (extended abstract). In: FOCS. pp. 162–167 (1986)