Secure Multiparty RAM Computation in Constant Rounds[⋆,⋆⋆]

Sanjam Garg[1], Divya Gupta[1], Peihan Miao[1], and Omkant Pandey[2]

[1] University of California, Berkeley
{sanjamg,divyagupta2016,peihan}@berkeley.edu
[2] Stony Brook University, omkant@gmail.com

Abstract. Secure computation of a random access machine (RAM) program typically entails that it be first converted into a circuit. This conversion is unimaginable in the context of big-data applications where the size of the circuit can be exponential in the running time of the original RAM program. Realizing these constructions, without relinquishing the efficiency of RAM programs, often poses considerable technical hurdles. Our understanding of these techniques in the multi-party setting is largely limited. Specifically, the round complexity of all known protocols grows linearly in the running time of the program being computed.

In this work, we consider the multi-party case and obtain the following results:

- *Semi-honest model*: We present a constant-round black-box secure computation protocol for RAM programs. This protocol is obtained by building on the new black-box garbled RAM construction by Garg, Lu, and Ostrovsky [FOCS 2015], and constant-round secure computation protocol for circuits of Beaver, Micali, and Rogaway [STOC 1990]. This construction allows execution of multiple programs on the same persistent database.
- *Malicious model*: Next, we show how to extend our semi-honest results to the malicious setting, while ensuring that the new protocol is still constant-round and black-box in nature.

1 Introduction

Alice, Bob, and Charlie jointly own a large private database D. For instance, the database D can be a concatenation of their individually owned private databases. They want to compute and learn the output of arbitrary dynamically chosen

[⋆] This paper was presented jointly with [25] in proceedings of the 14^{th} IACR Theory of Cryptography Conference (TCC) 2016-B.

[⋆⋆] Research supported in part from a DARPA/ARL SAFEWARE Award, AFOSR Award FA9550-15-1-0274, NSF CRII Award 1464397 and a research grant from the Okawa Foundation. The views expressed are those of the author and do not reflect the official policy or position of the funding agencies.

private random access machine (RAM) programs $P_1, P_2, \ldots$, on private inputs $x_1, x_2, \ldots$ and the previously stored database, which gets updated as these programs are executed. Can we do this?

Beginning with the seminal results of Yao [41] and Goldreich, Micali, and Wigderson [17], cryptographic primitives for secure computations are customarily devised for circuits. Using these approaches for random access machine (RAM) programs requires the conversion of the RAM program to a circuit. Using generic transformations [7,38], a program running in time T translates into a circuit of size $O(T^3 \log T)$. Additionally, the obtained circuit must grow at least with the size of the input that includes data, which can be prohibitive for various applications. In particular, this dependence on input length implies an exponential slowdown for binary search. For instance, in the example above, for each program that Alice, Bob, and Charlie want to compute, communication and computational complexities of the protocol need to grow with the size of the database. Using fully homomorphic encryption [13], one can reduce the communication complexity of this protocol, but not the computational cost, which would still grow with the size of the database. Therefore, it is paramount that we realize RAM friendly secure computation techniques, that do not suffer from these inefficiencies.

Secure computation for RAM programs. Motivated by the above considerations, various secure computation techniques that work directly for RAM programs have been developed. For instance, Ostrovsky and Shoup [36] achieve general secure RAM computation using oblivious RAM techniques [16,35,18]. Subsequently, Gordon et al. [20] demonstrate an efficient realization based on specialized number-theoretic protocols. However, all these and other follow-up works require round complexity linear in the running time of the program. This changed for the two-party setting with the recent results on garbled RAM [32,14,12] and its black-box variant [11].[3] However, these round-efficient results are limited to the two-party setting.

In this work, we are interested in studying this question in the multiparty setting in the following two natural settings of RAM computations: persistent database setting and the non-persistent database setting. Furthermore, we want constructions that make only a black-box use of the underlying cryptographic primitives.

Persistent vs. non-persistent database. In the setting of RAM programs, the ability to store a *persistent* private database that can be computed on multiple times can be very powerful. Traditionally, secure computation on RAM programs is thus studied in two models. In the first model, called the *persistent* database model, one considers execution of many programs on the same

[3] We note that several other cutting-edge results [19,15,6,4,31] have been obtained in non-interactive secure computation over RAM programs but they all need to make strong computational assumptions such as [39,10,9]. Additionally they make non-black-box use of the underlying cryptographic primitives.

database over a time period; the database can be modified by these programs during their execution and these changes persist over time. In this setting, the database can be huge and the execution time of each program does not need to depend on the size of the database.

In the non-persistent database setting, one considers only a single program execution. This setting is extremely useful in understanding the underlying difficulties in obtaining a secure solution.

Black-box vs. non-black-box. Starting with Impagliazzo-Rudich [27,26], researchers have been very interested in realizing cryptographic goals making just a black-box use of underlying primitive. It has been the topic of many important recent works in cryptography [29,37,40,21,23]. On the other hand, the problem of realizing black-box construction for various primitive is still open, e.g. multi-statement non-interactive zero-knowledge [5,8,24] and oblivious transfer extension [1].[4] From a complexity perspective, black-box constructions are very appealing as they often lead to conceptually simpler and qualitatively more efficient constructions.[5]

Note that putting together Garbled RAM construction of Garg, Lu, Ostrovsky, and Scafuro [12] and the multiparty secure computation protocol of Beaver, Micali, and Rogaway [2] immediately gives a non-black-box protocol for RAM programs with the persistent use of memory. However, motivated by black-box constructions and low round complexity, in this work, we ask:

Can we realize constant-round black-box *secure multiparty computation for RAM programs?*

1.1 Our Results

In this paper, addressing the above question, we obtain the first constant-round black-box protocols for both the semi-honest and the malicious setting. Specifically, we present the following results:

- **Semi-honest**: We show a constant-round black-box secure computation protocol for RAM programs. This protocol is obtained by building on the new black-box garbled RAM construction by Garg, Lu, and Ostrovsky [11], and constant round secure computation protocol for circuits of Beaver, Micali, and Rogaway [2]. Our construction allows for the execution of multiple programs on the same persistent database. In our construction, for an original database of size M, one party needs to maintain a persistent database of size $M \cdot \mathsf{poly}(\log M, \kappa)$. The communication and computational complexities of each program evaluation grow with $T \cdot \mathsf{poly}(\log T, \log M, \kappa)$ where T is the running time of the program and κ is the security parameter.

[4] Interestingly for oblivious transfer extension we do know black-box construction based on stronger assumptions [28].

[5] Additionally, black-box constructions enable implementations agnostic to the implementation of the underlying primitives. This offers greater flexibility allowing for many optimizations, scalability, and choice of implementation.

- **Malicious**: Next we enhance the security of our construction from semi-honest setting to malicious, while ensuring that the new protocol is still constant-round and black-box. In realizing this protocol we build on the constant round black-box secure computation protocol of Ishai, Prabhakaran, and Sahai [30]. However, this result is only for the setting of the non-persistent database.[6]

Both our constructions only make a black-box use of one-way functions in the OT-hybrid model.

1.2 Concurrent and Independent Work

In a concurrent and independent work, Hazay and Yanai [25] consider the question of malicious secure two-party secure RAM computation. They present a constant-round protocol building on the the semi-honest two-party protocols [14,12]. They achieve a similar result as ours in the two-party setting but make a non-black-box use of one-way functions. Moreover, they allow running of multiple programs on a persistent database when all the programs as well as the inputs are known beforehand to the garbler.[7]

An independent work of Miao [33] addresses the same problem as [25] but for the standard notion of persistent database that allows for programs and inputs of later executions to be chosen dynamically based on previous executions. [33] achieves a constant-round malicious secure protocol in the OT-hybrid with the use of random oracle building on the techniques of [3] and [34].

2 Our Techniques

Semi-honest setting with a single program. First, we consider the problem of constructing a semi-honest secure protocol for multi-party RAM computation. That is, consider n parties $Q_1, \ldots, Q_n$ and a database $D = D_1 || \ldots || D_n$ such that Q_i holds the database D_i. They want to securely evaluate a program P on input $x = x_1, \ldots, x_n$ w.r.t. the database D, where x_i is the secret input of party Q_i. Recall that our goal is to construct a constant-round protocol that only makes a black-box use of cryptographic primitives such as one-way function and oblivious transfer (OT). Moreover, we require that our protocol should only incur a one-time communication and computational cost that is linear in the size of D up to poly-logarithmic factors. Subsequently, evaluating each new program should require communication and computation that is only linear in the running time of that program up to poly-logarithmic factors.

[6] We elaborate on the fundamental issue in extending this result to the persistent database setting at the end of next section.

[7] We note that our malicious secure protocol also achieves this weaker notion of persistent database, but in this paper we only focus on the stronger notion of persistent data where later programs and inputs can be chosen dynamically.

High level approach. Our starting point would be garbled RAM that solves the problem in the two-party setting. Recall that a garbled RAM is the RAM analogue of Yao's garbled circuits [41], and allows for multiple program executions on a persistent database. Recently, Garg et al. [11] gave a construction of a garbled RAM that only makes a black-box use of one-way functions. Given this primitive, a natural approach would be to generate the garbled RAM via a secure computation protocol for circuits. However, since garbled RAM is a cryptographic object and builds on one-way functions, a straight-forward application of this approach leads to an immediate problem of non-black-box use of one-way functions.

Garbled RAM abstraction. To handle the above issue regarding non-black-box use of one-way functions, we would massage the garbled RAM construction of [11] such that all calls to one-way functions are performed locally by parties and ensure that the functionality computed by generic MPC is information-theoretic. Towards this goal, we need to understand the structure of the garbled RAM of [11] in more detail. Next, we abstract the garbled RAM of [11] and describe the key aspects of the construction, which avoids the details irrelevant for understanding our work.

The garbled RAM for memory database D, program P and input x consists of the garbled memory $\tilde{D}$, the garbled program $\tilde{P}$, and the garbled input $\tilde{x}$. At a high level, the garbled memory $\tilde{D}$ consists of a collection of memory garbled circuits (for reading and writing to the memory) that invoke other garbled circuits depending on the input and the execution. More precisely, the garbled circuits of the memory are connected in a specific manner and each garbled circuit has keys of several other garbled circuits hard-coded inside it and it outputs input labels for the garbled circuit that is to be executed next. Moreover, for some of these garbled circuits, labels for partial inputs are revealed at the time of the generation of garbled RAM, while the others are revealed at run-time. Among the labels revealed at generation time, semantics of some of the labels is public, while the semantics of the others depend on the contents of the database as well as the randomness of the ORAM used.[8] Similarly, the garbled program $\tilde{P}$ consists of a sequence of garbled circuits for CPU steps such that each circuit has input labels of several garbled circuits, from the memory and the program, hard-coded inside itself. Finally, the garbled input consists of the labels for the first circuit in the garbled program that are revealed depending on the input x.

Our crucial observation about [11] is the following: Though each circuit has hard-coded *secret* labels for several other garbled circuits, the overall structure of the garbled memory as well as garbled program is public. That is, how the garbled circuits are connected in memory and the program as well as the structure of hard-coding is fixed and public, independent of the actual database or the program being garbled. This observation would be useful in two aspects:

[8] The labels revealed at generation time are later referred to as the tabled garbled information. For security of garbled RAM, it is crucial to hide the semantics of the labels dependent on the database contents and we will revisit this later in the technical overview.

(1) To argue that the functionality being computed using the generic MPC for circuits is information theoretic. This is crucial in getting a black-box secure computation protocol for RAM. (2) When running more than one program on a persistent database, the basic structure of garbled RAM being public ensures that the cost of garbling additional programs does not grow with the size of the database.

Using above observations, we provide a further simplified formalization of the garbled RAM scheme of [11], where intuitively, we think of the circuits of memory and program as universal circuits that take secret hard-coded labels as additional inputs.[9] The labels for these additional input wires now have to be revealed at the time of garbled RAM generation. For details refer to Section 3.2. In light of this, our task is to devise a mechanism to generate all these garbled circuits and (partial) labels in a distributed manner securely. As mentioned above, since these garbled circuits use one-way functions (in generating encrypted gate tables), we cannot generate them naïvely.

Handling the issue of non-black-box use of one-way functions. We note that the garbled RAM of [11] makes a black-box use of a circuit garbling scheme and hence, can be instantiated using any secure circuit garbling scheme. This brings us to the next tool we use from the literature, which is the distributed garbling scheme of Beaver et al. [2], referred to as BMR in the following. In BMR, for each wire in the circuit, every party contributes a share of the label such that the wire-label is a concatenation of label shares from all the parties. Moreover, all calls to PRG (for generating encryptions of gate tables) are done locally such that given these PRG outputs and label shares, the generation of a garbled gate-table is information theoretic. This ensures that the final protocol of BMR is black-box in use of one-way functions. Our key observation is that we can instantiate the black-box garbled RAM of [11] with the BMR distributed garbling as the underlying circuit garbling scheme.

Based on what we have described so far, to obtain a constant-round semi-honest secure protocol for RAM, we would do the following: First, we would view the garbled RAM of [11] as a collection of suitable garbled circuits with additional input wires corresponding to the hardcoded secret labels (for simplicity). Next, we would use BMR distributed garbling as the underlying circuit garbling scheme, where each party computes the labels as well as PRG outputs locally. And, finally, we would run a constant-round black-box secure computation protocol for circuits (that is, BMR) to generate all the garbled circuits of the garbled RAM along with labels for partial inputs. In Section 5.3, we argue that the functionality being computed by MPC is information theoretic. Hence, this gives a black-box protocol.

Subtlety with use of ORAM. At first, it seems that we can generate all the garbled circuits of the garbled RAM in parallel via the MPC. But, separating the generation of garbled circuits creates a subtle problem in how garbled RAM

[9] Though this transformation is not crucial for security, it helps simplify the exposition of our protocol.

internally uses oblivious RAM. As mentioned before, some of the labels revealed at the time of garbled RAM generation depend on the database contents and the randomness used for ORAM. For security, the randomness of ORAM is contributed by all the parties and any sub-group of the parties does not learn the semantics of these labels. Therefore, separating the generation of garbled circuits requires all the parties to input the entire database to each garbled circuit generation, which would violate the efficiency requirements. In particular, efficiency of garbling the database would be at least quadratic in its size.

We solve this problem by bundling together the generation of all the garbled circuits under one big MPC. This does not harm security as well as provides the desired efficiency guarantees. More precisely, all the garbled circuits are generated by a single MPC protocol, where all the parties only need to input once the entire database along with all the randomness for the oblivious RAM (as well as their label shares and PRG outputs). We defer the details of this to the main body. There we also describe how we can extend this protocol for the setting of multiple program executions on a persistent database.

Malicious Setting. Next, we consider the case of malicious security. Again, to begin with, consider the case of a single program execution. For malicious security, we change the underlying secure computation protocol for generating garbled RAM to be malicious secure instead of just semi-honest secure. This would now ensure that each garbled circuit is generated correctly. Given that this secure computation is correct and malicious secure, the only thing that a malicious adversary can do is choose inputs to this protocol incorrectly or inconsistently. More precisely, as we will see in the main body, it is crucial that the PRG outputs fed into the secure computation protocol are correct. In fact, use of incorrect PRG values can cause honest parties to abort during evaluation of generated garbled RAM. This would be highly problematic for security if the adversary can cause input-dependent abort of honest parties as this is not allowed in ideal world. Note that we cannot use zero-knowledge proofs to ensure the correctness of PRG evaluations as this would lead to a non-black-box use of one-way functions. To get around this hurdle, we prove that the probability that an honest party aborts is independent of honest party inputs and depends only on the PRG values used by the adversary. In fact, given the labels as well as PRG outputs used by our adversary, our simulator can simulate which honest parties would abort and which honest parties would obtain the output.

The case of persistent data in the malicious setting. The final question is, can we extend the above approach to handle multiple programs? In the malicious setting, the adversary can choose the inputs for the second program based on the garbled memory that it has access to. Note that the garbled RAM of [11] does not guarantee security when the inputs can be chosen adaptively given the garbled RAM. Recall that the garbled memory of [11] consists of a collection of garbled circuits. In fact, to construct a scheme that satisfies this stronger security guarantee will require a circuit garbling scheme with the corresponding stronger security guarantee. In other words, we would need a circuit garbling scheme that is adaptively secure where the size of garbled input does not grow with the size of the circuit. However, we do not know of any such scheme in the standard model, i.e., without programmable random oracle assumption. Hence, we leave open the question of black-box malicious security for executing multiple RAM programs on a persistent database.

3 Preliminaries

We describe garbled RAM formally and give a brief overview of black box garbled RAM construction from [11]. Here we describe an abstraction of their construction which will suffice to describe our protocol for secure multi-party RAM computation as well as its security proof. Parts of this section have been taken verbatim from [14] and [11]. In the following, let κ be the security parameter. For a brief description of RAM model and garbled circuits, refer to the full version of this paper.

3.1 Garbled RAM

The garbled RAM [32,14,12] is the extension of garbled circuits to the setting of RAM programs. Here, the memory data D is garbled once and then many different garbled programs can be executed sequentially with the memory changes persisting from one execution to the next.

Definition 1. *A* secure single-program garbled RAM *scheme consists of four procedures* (GData, GProg, GInput, GEval) *with the following syntax:*

- $(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D)$*: Given a security parameter* 1^κ *and memory* $D \in \{0,1\}^M$ *as input,* GData *outputs the garbled memory* $\tilde{D}$ *and a key* s*.*
- $(\tilde{P}, s^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log M}, 1^t, P, s, m)$ *: Takes the description of a RAM program* P *with memory-size* M *and running-time* t *as input. It also requires a key* s *(produced by* GData*) and current time* m*. It then outputs a garbled program* $\tilde{P}$ *and an input-garbling-key* s^{in}*.*
- $\tilde{x} \leftarrow \mathsf{GInput}(1^\kappa, x, s^{in})$*: Takes as input* $x \in \{0,1\}^n$ *and an input-garbling-key* s^{in}*, and outputs a garbled-input* $\tilde{x}$*.*
- $(y, \tilde{D}') = \mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x})$*: Takes a garbled program* $\tilde{P}$*, garbled input* $\tilde{x}$ *and garbled memory data* $\tilde{D}$ *and outputs a value* y *along with updated garbled data* $\tilde{D}'$*. We model* GEval *itself as a RAM program that can read and write to arbitrary locations of its memory initially containing* $\tilde{D}$*.*

Efficiency: *The run-time of* GProg *and* GEval *are* $t \cdot \mathsf{poly}(\log M, \log T, \kappa)$*, which also serves as the bound on the size of the garbled program* $\tilde{P}$*. Here,* T *denotes the combined running time of all programs. Moreover, the run-time of* GData *is* $M \cdot \mathsf{poly}(\log M, \log T, \kappa)$*, which also serves as an upper bound on the size of* $\tilde{D}$*. Finally the running time of* GInput *is* $n \cdot \mathsf{poly}(\kappa)$*.*

Correctness: *For correctness, we require that for any initial memory data* $D \in \{0,1\}^M$ *and any sequence of programs and inputs* $\{P_i, x_i\}_{i \in [\ell]}$*, following* *holds: Denote by y_i the output produced and by D_{i+1} the modified data resulting from running $P_i(D_i, x_i)$. Let $(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D)$. Also, let $(\tilde{P}_i, s_i^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log M}, 1^{t_i}, P_i, s, \sum_{j\in[i-1]} t_j)$, $\tilde{x}_i \leftarrow \mathsf{GInput}(1^\kappa, x_i, s_i^{in})$ and $(y_i', \tilde{D}_{i+1}) = \mathsf{GEval}^{\tilde{D}_i}(\tilde{P}_i, \tilde{x}_i)$. Then, $\Pr[y_i = y_i', \quad \text{for all } i \in \{1, \ldots, \ell\}] = 1$.*

Security: *For security, we require that there exists a PPT simulator* GramSim *such that for any initial memory data $D \in \{0,1\}^M$ and any sequence of programs and inputs $\{P_i, x_i\}_{i\in[\ell]}$, the following holds: Denote by y_i the output produced and by D_{i+1} the modified data resulting from running $P_i(D_i, x_i)$. Let $(\tilde{D}, s) \leftarrow \mathsf{GData}(1^\kappa, D)$, $(\tilde{P}_i, s_i^{in}) \leftarrow \mathsf{GProg}(1^\kappa, 1^{\log M}, 1^{t_i}, P_i, s, \sum_{j\in[i-1]} t_j)$ and $\tilde{x}_i \leftarrow \mathsf{GInput}(1^\kappa, x_i, s_i^{in})$, then*

$$(\tilde{D}, \{\tilde{P}_i, \tilde{x}_i\}_{i\in[\ell]}) \overset{\text{comp}}{\approx} \mathsf{GramSim}(1^\kappa, 1^M, \{1^{t_i}, y_i\}_{i\in[\ell]})$$

3.2 Black-Box Garbled RAM of [11]

The work of [11] gives a construction of garbled RAM that only makes a black-box use of one-way functions. In particular, it proves the following theorem.

Theorem 1 ([11]). *Assuming the existence of one-way functions, there exists a secure black-box garbled RAM scheme for arbitrary RAM programs satisfying the efficiency, correctness and security properties stated in Definition 1.*

Below, we describe the construction of [11] at a high level. We describe the algorithms $(\mathsf{GData}, \mathsf{GProg}, \mathsf{GInput})$. In following, in the context of garbled circuits, *labels* refers to one of the labels for an input bit and *keys* refers to both labels (one for 0 and one for 1) corresponding to an input bit.

[11] construct black-box garbled RAM in two steps. First a garbled RAM scheme is constructed under the weaker security requirement of unprotected memory access (UMA2-security) where only the sequence of memory locations being accessed is revealed. Everything else about the program, data and the input is hidden. Next this weaker security guarantee is amplified to get full security by using statistical oblivious RAM that hides the memory locations being accessed.

Garbled RAM achieving UMA2-security Let the corresponding procedures be $(\widehat{\mathsf{GData}}, \widehat{\mathsf{GProg}}, \widehat{\mathsf{GInput}}, \widehat{\mathsf{GEval}})$.

- $(\tilde{D}, s) \leftarrow \widehat{\mathsf{GData}}(1^\kappa, D)$: $\tilde{D}$ consists of a collection of garbled circuits and a tabled garbled information Tab. The key s corresponds to a PRF key.

  Garbled Circuits. The collection of garbled circuits is organized as a binary tree of depth $d = O(\log |D|)$ and each node consists of a sequence of *garbled circuits.*[10] For any garbled circuit, its *successor* (resp. predecessor) is defined

[10] Note that for security, any garbled circuit can only be executed once. Hence, to enable multiple reads from the memory, each node consists of a sequence of garbled

to be the next (resp. previous) node in the sequence. For a garbled circuit, all garbled circuits in parent (resp. children) node are called parents (resp. children). There are two kinds of circuits: leaf circuits $\mathsf{C}^{\mathsf{leaf}}$ (at the leaves of the tree) and non-leaf circuits $\mathsf{C}^{\mathsf{node}}$ (at the internal nodes of the tree). Intuitively speaking, the leaf nodes carry the actual data.

Each garbled circuit has hard-coded inside it (partial) keys of a set of other garbled circuits. We emphasize that for any circuit C, the keys that are hard-coded inside it are fixed (depending on the size of data) and is independent of the actual contents of the data. This would be crucial later.

Tabled garbled information. For each node in the tree as described above, the garbled memory consists of a table of information $\mathsf{Tab}(i, j)$, where (i, j) denotes the j^{th} node at i^{th} level from the root. Note that d denotes the depth of the tree. The tabulated information $\mathsf{Tab}(i, j)$ contains labels for partial inputs of the first garbled circuit in the sequence of circuits at node (i, j) (i.e., one label for some of the input bits). As the garbled memory is consumed by executing the garbled circuits, the invariant is maintained that the tabulated information contains partial labels for the first unused garbled circuit at that node.

A crucial point to note is the following: Labels in Tab entry corresponding to non-leaf nodes, i.e., $\mathsf{C}^{\mathsf{node}}$, depend on the keys on some other garbled circuit. Also, the tabulated information for the leaf nodes depends on actual value in the data being garbled. More precisely, the entry $\mathsf{Tab}(d, j)$ for level d of the leaves contains the partial labels for the first $\mathsf{C}^{\mathsf{leaf}}$ circuit at j^{th} leaf corresponding to value $D[j]$ value.[11]

The keys of the all the garbled circuits are picked to be outputs of a PRF with key s on appropriate inputs.

- $(\tilde{P}, s^{in}) \leftarrow \widehat{\mathsf{GProg}}(1^\kappa, 1^{\log M}, 1^t, P, s, m)$: The garbled program $\tilde{P}$ consists of a sequence of garbled circuits, called $\mathsf{C}^{\mathsf{step}}$. Again, each garbled circuit has (partial) keys of some circuits of $\tilde{P}$ and $\tilde{D}$ hard-coded inside it. We emphasize that for any circuit $\mathsf{C}^{\mathsf{step}} \in \tilde{P}$, which keys are hard-coded inside it is fixed and is independent of the actual program and actual data.

  s^{in} corresponds to the keys of the first circuit in this sequence.
- $\tilde{x} \leftarrow \widehat{\mathsf{GInput}}(1^\kappa, x, s^{in})$: The GInput algorithm uses x as selection bits for the keys provided in s^{in}, and outputs $\tilde{x}$ that is just the selected labels.

Remark 1. Note that for [11] the labels in Tab for $\mathsf{C}^{\mathsf{node}}$ and hard-coded key values are independent of the actual data D and input x. The labels in Tab for $\mathsf{C}^{\mathsf{leaf}}$ depend on data D and labels in $\tilde{x}$ depend on input x.

circuits. Number of garbled circuits at any node is chosen carefully. See [11] for details. For our purpose, we do not need to specify the number of garbled circuits at each node, but it is worth emphasizing that the total number of garbled circuits is $|D| \cdot \mathsf{poly}(\log |D|, \kappa)$.

[11] Note that it is public that j^{th} leaf corresponds to $D[j]$. Later, statistical ORAM is used to hide this correspondence.

Garbled RAM achieving full security [11] prove the following lemma.

Lemma 1. *[11] Given a UMA2-secure garbled RAM scheme* $(\widehat{\mathsf{GData}}, \widehat{\mathsf{GProg}}, \widehat{\mathsf{GInput}}, \widehat{\mathsf{GEval}})$ *for programs with (leveled) uniform memory access, and a statistical ORAM scheme* $(\mathsf{OData}, \mathsf{OProg})$ *giving (leveled) uniform memory access that protects the access pattern, there exists a fully secure garbled RAM scheme.*

The construction works by first applying ORAM compiler followed by UMA2-secure garbled RAM compiler. More formally,

- $\mathsf{GData}(1^\kappa, D)$: Execute $(D^*) \leftarrow \mathsf{OData}(1^\kappa, D)$ followed by $(\tilde{D}, s) \leftarrow \widehat{\mathsf{GData}}(1^\kappa, D^*)$. Output $(\tilde{D}, s)$. Note that OData does not require a key as it is a statistical scheme.
- $\mathsf{GProg}(1^\kappa, 1^{\log M}, 1^t, P, \widehat{s}, m)$: Execute $P^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log M}, 1^t, P)$ followed by $(\tilde{P}, s^{in}) \leftarrow \widehat{\mathsf{GProg}}(1^\kappa, 1^{\log M'}, 1^{t'}, P^*, s, m)$. Output $(\tilde{P}, s^{in})$.
- $\mathsf{GInput}(1^\kappa, x, s^{in})$: Note that x is valid input for P^*. Execute $\tilde{x} \leftarrow \widehat{\mathsf{GInput}}(1^\kappa, x, s^{in})$, and output $\tilde{x}$.
- $\mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x})$: Execute $y \leftarrow \widehat{\mathsf{GEval}}^{\tilde{D}}(\tilde{P}, \tilde{x})$ and output y.

Note that UMA-2-secure garbled RAM does not hide which leaf nodes of garbled data correspond to which bit of data D. In the garbled RAM with full security, this is being hidden due to compilation by ORAM. In the final garbled RAM with full security, which keys are hardwired in each garbled circuit remains public as before. Also, which keys are stored in Tab are public except those for $\mathsf{C}^{\mathsf{leaf}}$ as they correspond to data values. These are determined by the randomness used in ORAM as well as actual data.

Transformation of garbled RAM to remove the hard-coding of keys. As is clear from the above description, the garbled RAM $(\tilde{D}, \tilde{P}, \tilde{x})$ consists of a collection of garbled circuits of three types: $\mathsf{C}^{\mathsf{leaf}}, \mathsf{C}^{\mathsf{node}}$ and $\mathsf{C}^{\mathsf{step}}$ and a collection of labels given in tabled garbled information of $\tilde{D}$ and $\tilde{x}$. Each of these circuits have (partial) keys of other garbled circuits hard-coded inside them and this structure of hard-coding is public and fixed independent of the actual data, program and randomness of ORAM. In this work, we change the circuits in garbled RAM construction as follows: We consider these hard-coded values as additional inputs whose labels are given out at time of garbled RAM generation. Once we remove the hard-coding of keys from inside these circuits, the structure of these circuits is public and known to all. The remark below summarizes our garbled RAM abstraction.

Remark 2. **Garbled RAM abstraction.** The garbled RAM consists of a collection of garbled circuits. The structure of these garbled circuits as well as semantics of each of the input wires are public. For each of these garbled circuits, labels for partial inputs are revealed at the time of garbled RAM generation. Labels which are not revealed become known while evaluating the garbled RAM.

Labels which are revealed correspond to one of the following inputs: The ones for which labels were given in Tab, earlier hardcoded keys, or the input x.

The semantics of the labels revealed in Tab corresponding to $\mathsf{C}^{\mathsf{node}}$ is public. But, ORAM hides the semantics of the labels in Tab corresponding to $\mathsf{C}^{\mathsf{leaf}}$ and these depend on data values in memory at specific locations. Moreover, the mapping of these leaves to locations of memory is also hidden by ORAM and is determined by the randomness used by ORAM.

It is easy to see that the garbed RAM obtained after this transformation is equivalent to the original garbled RAM of [11]. In the following, the garbled RAM will refer to this simpler version of garbled RAM where circuits do not have any hard-coded keys. This transformation would help in ensuring that the secure computation protocols that we describe only make a black-box use of cryptography.

4 Our Model

In this section we define the security of secure computation for RAM programs for the case of persistent database. In this work, we consider both semi-honest as well as malicious adversaries. A semi-honest adversary is guaranteed to follow the protocol whereas a malicious adversary may deviate arbitrarily from the protocol specification. In particular, a malicious adversary may refuse to participate in the protocol, may use an input that is different from prescribed input, and may be abort prematurely blocking the honest parties from getting the output. We consider static corruption model, where the adversary may corrupt an arbitrary collection of the parties, and this set is fixed before the start of the protocol. We consider *security with abort* using real world and ideal world simulation based definition. We consider a ideal computation using incorruptible third party to whom parties send their inputs and receive outputs. This ideal scenario is secure by definition. For security, we require that whatever harm the adversary can do by interacting in the real protocol is mimicked by an ideal world adversary. We provide a detailed formal description of our model in our full version.

Consider parties $Q_1, \ldots, Q_n$ holding secret database $D_1, \ldots, D_n$, respectively. They want to compute a sequence of programs $\boldsymbol{P} = (P^{(1)}, \ldots, P^{(\ell)})$ on the persistent database $D = D_1 || \ldots || D_n$. Q_i has secret input $x_i^{(j)}$ for program $P^{(j)}$.

The overall output of the ideal-world experiment consists of all the values output by all parties at the end, and is denoted by $\mathsf{Ideal}_{\mathcal{S}}^{\boldsymbol{P}}(1^\kappa, D, \{\boldsymbol{x}^{(j)}\}_{j \in [\ell]}, z)$, where z is the auxiliary input given to the ideal adversary $\mathcal{S}$ at the beginning. In the case of an semi-honest adversary $\mathcal{S}$, all parties receive the same output from the trusted functionality. In the case of the malicious adversary, $\mathcal{S}$ after receiving the output from the trusted party, chooses a subset of honest parties $\mathcal{J} \subseteq [n] \setminus \mathcal{I}$ and sends $\mathcal{J}$ to the trusted functionality. Here, $\mathcal{I} \subseteq [n]$ denotes the set of corrupt parties. The trusted party sends the output to parties in $\mathcal{J}$ and special symbol $\perp$ to all the other honest parties.

Similarly, the overall output of the real-world experiment consists of all the values output by all parties at the end of the protocol, and is denoted by

$\mathsf{Real}^{\pi}_{\mathcal{A}}(1^{\kappa}, D, \{\boldsymbol{x}^{(j)}\}_{j\in[\ell]}, z)$, where z is the auxiliary input given to real world adversary $\mathcal{A}$. Then, security is defined as follows:

Definition 2 (Security). *Let $\boldsymbol{P} = (P^{(1)}, \ldots, P^{(\ell)})$ be a sequence of well-formed RAM programs and let π be a n-party protocol for $\boldsymbol{P}$. We say that π securely computes $\boldsymbol{P}$, if for every $\{D_i, x_i^{(1)}, \ldots, x_i^{(\ell)}\}_{i\in[n]}$, every auxiliary input z, every real world adversary $\mathcal{A}$, there exists an ideal world $\mathcal{S}$ such that $\mathsf{Real}^{\pi}_{\mathcal{A}}(1^{\kappa}, D, \{\boldsymbol{x}^{(j)}\}_{j\in[\ell]}, z) \approx \mathsf{Ideal}^{\boldsymbol{P}}_{\mathcal{S}}(1^{\kappa}, D, \{\boldsymbol{x}^{(j)}\}_{j\in[\ell]}, z)$.*

Efficiency: We also want the following efficiency guarantees. We consider the following two natural scenarios: Below, M is the size of total database, i.e. $M = |D|$, t_j denotes the running time of $P^{(j)}$ and $T = \max_j t_j$

1. **All parties do computation:** In this case, for all the parties we require the total communication complexity and computation complexity to be bounded by $M \cdot \mathsf{poly}(\log M, \log T, \kappa) + \sum_{j\in[\ell]} t_j \cdot \mathsf{poly}(\log M, \log t_j, \kappa)$ and each party needs to store total database and program of size at most $M \cdot \mathsf{poly}(\log M, \log T, \kappa) + T \cdot \mathsf{poly}(\log M, \log T, \kappa)$. With each additional program to be computed, the additional communication complexity should be $t_j \cdot \mathsf{poly}(\log M, \log t_j, \kappa)$.
2. **Only one party does the computation:** In this case, as before the communication complexity and computation complexity of the protocol is bounded by $M \cdot \mathsf{poly}(\log M, \log T, \kappa) + \sum_{j\in[\ell]} t_j \cdot \mathsf{poly}(\log M, \log t_j, \kappa)$. But, in some cases such as semi-honest security, we can optimize on the space requirements of the parties and all parties do not require space proportional to the total database. The one designated party who does the computation needs to store $M \cdot \mathsf{poly}(\log M, \log T, \kappa) + T \cdot \mathsf{poly}(\log M, \log T, \kappa)$. All the other parties Q_i only need to store their database of size $|D_i|$. [12]

5 Semi-Honest Multi-party RAM Computation

In this section, we describe the semi-honest secure protocol for RAM computation. We prove the following theorem.

Theorem 2. *There exists a constant-round semi-honest secure multiparty protocol for secure RAM computation for the case of persistent database in the OT-hybrid model that makes a black-box use of one-way functions. This protocol satisfies the security and the efficiency requirements of Section 4.*

Below, we first describe a secure protocol for the case of a single program execution for the case when all the parties compute the program and hence, need space proportional to the total size of the database. Later, we describe how our

[12] During the protocol execution all parties would need space proportional to $M \cdot \mathsf{poly}(\log M, \log T, \kappa) + T \cdot \mathsf{poly}(\log M, \log T, \kappa)$. Looking ahead, this is needed to run a protocol which outputs the garbled RAM to the designated party.

protocol can be extended to the case of multiple programs and optimizations of load balancing.

Protocol Overview. Our goal is to construct a semi-honest secure protocol for RAM computation. At a high level, in our protocol, the parties will run a multiparty protocol (for circuits) to generate the garbled RAM. We want a constant round protocol for secure computation that only makes a black-box use of one-way functions in the OT-hybrid model. Such a protocol was given by [2]. As already mentioned in technical overview, a naïve use of this protocol to generate the garbled RAM results in a non-black-box use of one way functions. The reason is the following: As explained before, the black-box garbled RAM of [11] consists of a collection of garbled circuits and hence, uses one-way functions inside it. Our main idea is to transform the garbled RAM of [11] in a way that allows each party to compute the one-way functions locally so that the functionality we compute using [2] is non-cryptographic (or, information theoretic). To achieve this, we again use ideas from distributed garbling scheme of [2]. Below we review their main result and the underlying garbling technique.

5.1 Distributed Garbling Protocol of [2]

Following result was proven by [2].

Theorem 3 ([2]). *There exists a constant-round semi-honest secure protocol for secure compuatation for circuits, which makes black-box calls to PRG in the OT-hybrid model. Let the protocol for a functionality $\mathcal{F}$ be denoted by $\Pi_{\mathsf{bmr}}^{\mathcal{F}}$ and the corresponding simulator be $\mathsf{Sim}_{\mathsf{bmr}}$.*

We describe this protocol next at a high level. Some of the following text has been taken from [22].

Suppose there are n parties $Q_1, \ldots, Q_n$ with inputs $x_1, \ldots, x_n$. The goal is the following: For any circuit C, at the end of the garbling protocol, each party holds a garbled circuit $\tilde{C}$ corresponding to C and garbled input $\tilde{x}$ corresponding to $x = x_1, \ldots, x_n$. Then, each party can compute $\tilde{C}$ on $\tilde{x}$ locally. At a high level, the evaluation is as follows: Recall that in a garbled circuit, each wire w has two keys $\mathsf{key}_0^{\mathsf{w}}$ and $\mathsf{key}_1^{\mathsf{w}}$: one corresponding to the bit being 0 and another corresponding to the bit being 1. In the multiparty setting, each wire also has a *wire mask* λ^w that determines the correspondence between the two wire keys and the bit value. More precisely, the key $\mathsf{key}_{\mathsf{b}}^{\mathsf{w}}$ corresponds to the bit $b \oplus \lambda^w$.

In the following, let F be a PRF and G be a PRG. The garbling protocol $\Pi_{\mathsf{bmr}}^{\mathsf{sh}}$ is as follows:

- **Stage 1.** Party Q_i picks a seed s_i for the PRF F. It generates its shares for keys for all the wires of C and wire masks as follows: Define $(\mathsf{k}_0^{\mathsf{w}}(\mathsf{i}), \mathsf{k}_1^{\mathsf{w}}(\mathsf{i}), \lambda_i^w) = F_{s_i}(w)$. In the final garbled circuit $\tilde{C}$, key for any wire w will be a concatenation of keys from all the parties. That is, $\mathsf{key}_{\mathsf{b}}^{\mathsf{w}} = \mathsf{k}_{\mathsf{b}}^{\mathsf{w}}(1) \circ \ldots \circ \mathsf{k}_{\mathsf{b}}^{\mathsf{w}}(\mathsf{n})$ and $\lambda^w = \lambda_1^w \oplus \ldots \oplus \lambda_n^w$.

- **Stage 2.** Recall that in a garbled circuit, for any gate, the keys for the output wires are encrypted under the input keys for all the four possible values for the inputs. These encryptions are stored in a garbled table corresponding to each gate. For all garbled circuit constructions, this step of symmetric encryption involves the use of one-way functions. In order to ensure black-box use of one-way functions, each party will make the PRG call locally. The parties locally expand their key parts into large strings that will be used as one-time pads to encrypt the key for the output wire labels. More precisely, Q_i expands the key parts $\mathsf{k}_0^{\mathsf{w}}(\mathsf{i})$ and $\mathsf{k}_1^{\mathsf{w}}(\mathsf{i})$ using PRG G to obtain two new strings, i.e., $(\mathsf{p}_\mathsf{b}^{\mathsf{w}}(\mathsf{i}), \mathsf{q}_\mathsf{b}^{\mathsf{w}}(\mathsf{i})) = G(\mathsf{k}_\mathsf{b}^{\mathsf{w}}(\mathsf{i}))$, for $b \in \{0,1\}$. Both $\mathsf{p}_\mathsf{b}^{\mathsf{w}}(\mathsf{i})$ and $\mathsf{q}_\mathsf{b}^{\mathsf{w}}(\mathsf{i})$ have length $n|\mathsf{k}_\mathsf{b}^{\mathsf{w}}(\mathsf{i})| = |\mathsf{key}_\mathsf{b}^{\mathsf{w}}|$ (enough to encrypt the key for output wire).
  More precisely, for every gate in C, a gate table is defined as follows: Let α, β be the two input wires and γ be the output wire, and denote the gate operation by $\otimes$. Party Q_i holds the inputs $\mathsf{p}_\mathsf{b}^{\alpha}(\mathsf{i}), \mathsf{q}_\mathsf{b}^{\alpha}(\mathsf{i}), \mathsf{p}_\mathsf{b}^{\beta}(\mathsf{i}), \mathsf{q}_\mathsf{b}^{\beta}(\mathsf{i})$ for $b \in \{0,1\}$ along with shares of masks $\lambda_i^\alpha, \lambda_i^\beta, \lambda_i^\gamma$. The garbled gate table is the following four encryptions:

$$\begin{aligned}
A_g &= \mathsf{p}_0^{\alpha}(1) \oplus \ldots \oplus \mathsf{p}_0^{\alpha}(\mathsf{n}) \oplus \mathsf{p}_0^{\beta}(1) \oplus \ldots \oplus \mathsf{p}_0^{\beta}(\mathsf{n}) \\
&\oplus \begin{cases} \mathsf{k}_0^{\gamma}(1) \circ \ldots \circ \mathsf{k}_0^{\gamma}(\mathsf{n}) \text{ if } \lambda^\alpha \otimes \lambda^\beta = \lambda^\gamma \\ \mathsf{k}_1^{\gamma}(1) \circ \ldots \circ \mathsf{k}_1^{\gamma}(\mathsf{n}) \text{ otherwise} \end{cases} \\
B_g &= \mathsf{q}_0^{\alpha}(1) \oplus \ldots \oplus \mathsf{q}_0^{\alpha}(\mathsf{n}) \oplus \mathsf{p}_1^{\beta}(1) \oplus \ldots \oplus \mathsf{p}_1^{\beta}(\mathsf{n}) \\
&\oplus \begin{cases} \mathsf{k}_0^{\gamma}(1) \circ \ldots \circ \mathsf{k}_0^{\gamma}(\mathsf{n}) \text{ if } \lambda^\alpha \otimes \overline{\lambda^\beta} = \lambda^\gamma \\ \mathsf{k}_1^{\gamma}(1) \circ \ldots \circ \mathsf{k}_1^{\gamma}(\mathsf{n}) \text{ otherwise} \end{cases} \\
C_g &= \mathsf{p}_1^{\alpha}(1) \oplus \ldots \oplus \mathsf{p}_1^{\alpha}(\mathsf{n}) \oplus \mathsf{q}_0^{\beta}(1) \oplus \ldots \oplus \mathsf{q}_0^{\beta}(\mathsf{n}) \\
&\oplus \begin{cases} \mathsf{k}_0^{\gamma}(1) \circ \ldots \circ \mathsf{k}_0^{\gamma}(\mathsf{n}) \text{ if } \overline{\lambda^\alpha} \otimes \lambda^\beta = \lambda^\gamma \\ \mathsf{k}_1^{\gamma}(1) \circ \ldots \circ \mathsf{k}_1^{\gamma}(\mathsf{n}) \text{ otherwise} \end{cases} \\
D_g &= \mathsf{q}_1^{\alpha}(1) \oplus \ldots \oplus \mathsf{q}_1^{\alpha}(\mathsf{n}) \oplus \mathsf{q}_1^{\beta}(1) \oplus \ldots \oplus \mathsf{q}_1^{\beta}(\mathsf{n}) \\
&\oplus \begin{cases} \mathsf{k}_0^{\gamma}(1) \circ \ldots \circ \mathsf{k}_0^{\gamma}(\mathsf{n}) \text{ if } \overline{\lambda^\alpha} \otimes \overline{\lambda^\beta} = \lambda^\gamma \\ \mathsf{k}_1^{\gamma}(1) \circ \ldots \circ \mathsf{k}_1^{\gamma}(\mathsf{n}) \text{ otherwise} \end{cases}
\end{aligned}$$

  In [2] this garbled table is generated by running a semi-honest secure computation protocol by the parties $Q_1, \ldots, Q_n$. Here, for the secure computation protocol, the private input of party Q_i are $\mathsf{p}_\mathsf{b}^{\alpha}(\mathsf{i}), \mathsf{q}_\mathsf{b}^{\alpha}(\mathsf{i}), \mathsf{p}_\mathsf{b}^{\beta}(\mathsf{i}), \mathsf{q}_\mathsf{b}^{\beta}(\mathsf{i}), \mathsf{k}_\mathsf{b}^{\gamma}(\mathsf{i}), \lambda_i^\alpha, \lambda_i^\beta, \lambda_i^\gamma$ for $b \in \{0,1\}$. Note that the garbled table is an information theoretic (or, non-cryptographic) function of the private inputs. Hence, the overall protocol is information theoretic in the OT-hybrid model. Moreover, to get the constant round result of [2], it was crucial that the garbled table generation circuit has depth constant (in particular, 2).
- **Stage 3.** The parties also get the garbled input $\tilde{x}$. For a wire w with value x_w, let $\varLambda^w = x_w \oplus \lambda_1^w \oplus \ldots \oplus \lambda_n^w$. All parties get $\varLambda^w, \mathsf{key}^{\mathsf{w}}_{\Lambda^\mathsf{w}}$. Parties also reveal their masks for each output wire λ_i^o.
- **Stage 4.** Finally, given the garbled circuit $\tilde{C}$ consisting of all the garbled tables and garbled input $\tilde{x}$, the parties can compute locally as follows: For any wire w, ρ^w denote its correct value during evaluation. It is maintained that for any wire w, each party learns the masked value $\varLambda^w$ and the label $\mathsf{key}^{\mathsf{w}}_{\Lambda^\mathsf{w}}$ where $\varLambda^w = \lambda^w \oplus \rho^w$. It is clearly true for the input wires. Now,

for any gate g with input wires α, β, each party knows $\Lambda^{\alpha}, \mathsf{key}^{\alpha}_{\Lambda^{\alpha}}, \Lambda^{\beta}, \mathsf{key}^{\beta}_{\Lambda^{\beta}}$. If $(\Lambda^{\alpha}, \Lambda^{\beta}) = (0, 0)$, decrypt the first row of the garbled gate, i.e., A_g, if $(\Lambda^{\alpha}, \Lambda^{\beta}) = (0, 1)$ decrypt B_g, if $(\Lambda^{\alpha}, \Lambda^{\beta}) = (1, 0)$ decrypt C_g, and else if $(\Lambda^{\alpha}, \Lambda^{\beta}) = (1, 1)$ decrypt D_g and obtain $\mathsf{key}^{\gamma} = \mathsf{k}^{\gamma}(1) \circ \ldots \circ \mathsf{k}^{\gamma}(\mathsf{n})$. Now, party Q_i checks the following: If $\mathsf{k}^{\gamma}(\mathsf{i}) = \mathsf{k}^{\gamma}_{\mathsf{b}}(\mathsf{i})$ for some $b \in \{0, 1\}$, it sets $\Lambda^{\gamma} = b$. Else, party Q_i aborts. Finally, each parties computes the output using λ^{o} and Λ^{o} for the output wires.

5.2 Garbled RAM Instantiated with Distributed Garbling of BMR

The aforementioned garbling protocol implies a special distributed circuit garbling scheme, which we refer to in the following as BMR scheme, denoted by $(\mathsf{GCircuit}_{\mathsf{bmr}}, \mathsf{Eval}_{\mathsf{bmr}}, \mathsf{CircSim}_{\mathsf{bmr}})$. It has the same syntax as the a secure circuit garbling scheme, but with the special labeling structure described above. The scheme has the following properties.

Black-box use of OWFs. The scheme only involves a black-box use of one-way functions in the OT-hybrid model.

Security. Since the above protocol from [2] is a semi-honest secure computation protocol, the BMR scheme is a secure circuit garbling scheme. That is, it does not reveal anything beyond the output of the circuit C on x to an adversary corrupting a set of parties $\mathcal{I} \subset [n]$. More precisely, we can abstract out the BMR scheme as well as its security as follows: Let us denote the collection of labels used by party Q_i using PRF s_i by Labels_i and the set of wire masks by λ_i. Similarly, let $\mathsf{Labels}_{\mathcal{I}}$ denote $\{\mathsf{Labels}_i\}_{i \in [\mathcal{I}]}$ and $\lambda_{\mathcal{I}}$ denote $\{\lambda_i\}_{i \in [\mathcal{I}]}$. Then, the following lemma states the security of the BMR scheme.

Lemma 2 (Security of BMR garbling scheme). *There exists a PPT simulator* $\mathsf{CircSim}_{\mathsf{bmr}}$ *such that* $\mathsf{CircSim}_{\mathsf{bmr}}(1^{\kappa}, C, x_{\mathcal{I}}, \mathsf{Labels}_{\mathcal{I}}, \lambda_{\mathcal{I}}, y) \approx (\tilde{C}, \tilde{x})$. *Here* $(\tilde{C}, \tilde{x})$ *correspond to the garbled circuit and the garbled input produced in the real world using* $\Pi^{\mathsf{sh}}_{\mathsf{bmr}}$ *conditioned on* $(\mathsf{Labels}_{\mathcal{I}}, \lambda_{\mathcal{I}})$. *We denote it by* $\mathsf{GCircuit}_{\mathsf{bmr}}(1^{\kappa}, C, x)\Big|_{(\mathsf{Labels}_{\mathcal{I}}, \lambda_{\mathcal{I}})}$.

The proof of the above lemma follows from the security of [2].

Distributed garbling scheme of garbled RAM. Our next step is instantiating the garbled RAM of [11] with the BMR circuit garbling scheme. As mentioned in Lemma 2, the BMR scheme $(\mathsf{GCircuit}_{\mathsf{bmr}}, \mathsf{Eval}_{\mathsf{bmr}}, \mathsf{CircSim}_{\mathsf{bmr}})$ is a secure circuit garbling scheme. And we note that [11] makes a black-box use of a secure circuit garbling scheme $(\mathsf{GCircuit}, \mathsf{Eval}, \mathsf{CircSim})$. Our key observation is that it can be instantiated using the BMR scheme. This would be very useful for our protocol of secure computation for RAM programs. When we instantiate the garbled RAM of [11] with the BMR scheme, the following lemma summarizes the security of the resulting garbled RAM relying upon Lemma 2.

Lemma 3 (Garbled RAM security with BMR garbling). *Instantiating the garbled RAM construction of [11] with the BMR circuit garbling scheme*

$(\mathsf{GCircuit}_{\mathsf{bmr}}, \mathsf{Eval}_{\mathsf{bmr}}, \mathsf{CircSim}_{\mathsf{bmr}})$ *gives a secure garbled RAM scheme. In particular, the garbler picks* $s_1, \ldots, s_n$ *as the seeds of the PRF to generate the keys of the garbled circuit. Let the* i^{th} *set of keys be* Labels_i*. Denote the resulting scheme by* $\mathsf{Gram}_{\mathsf{bmr}}$*. Denote the corresponding simulator for garbled RAM by* $\mathsf{GramSim}_{\mathsf{bmr}}$*, which would internally use* $\mathsf{CircSim}_{\mathsf{bmr}}$*. Using the security of garbled RAM and the security of BMR scheme, we have the following:*

$$\mathsf{Gram}_{\mathsf{bmr}}(1^\kappa, 1^t, D, P, x) \approx_c \mathsf{GramSim}_{\mathsf{bmr}}(1^\kappa, 1^t, 1^{|D|}, y),$$

where y *denotes the output of* $P(D, x)$*. In fact, using the security property of* $\mathsf{CircSim}_{\mathsf{bmr}}$*, we have the following stronger security property. Let* $(x = x_1, \ldots, x_n)$ *and* $D = (D_1 || \ldots || D_n)$*. Let* $\mathcal{I} \subset [n]$*. Then*

$$\mathsf{Gram}_{\mathsf{bmr}}(1^\kappa, 1^t, D, P, x, \mathsf{Labels}_{\mathcal{I}}, \mathcal{I}) \approx_c \mathsf{GramSim}_{\mathsf{bmr}}(1^\kappa, 1^t, 1^{|D|}, (x_{\mathcal{I}}, D_{\mathcal{I}}, \mathsf{Labels}_{\mathcal{I}}), y).$$

Recall that a garbled RAM consists of a collection of garbled circuits along with partial labels. It is easy to see that using the BMR garbling scheme preserves the black-box nature of the garbled RAM construction of [11].

Removing the hard-coding of keys in the scheme of [11]. As mentioned before, the garbled circuits in garbled RAM of [11] contain hard-coding of keys of other garbled circuits. For ease of exposition, we remove this hard-coding of sensitive information and provide the previously these values as additional inputs.[13] Moreover, labels corresponding to these new inputs would be revealed at the time of garbled RAM generation. More precisely, we would do the following:

Consider a circuit C in the original scheme of [11] which has the partial keys of some circuit C' hardcoded inside it. Since we will be using the BMR garbling scheme, key $\mathsf{key}_{\mathsf{b}}^{\mathsf{w}}$ of any wire w of C' consists of a concatenation of keys $\mathsf{k}_{\mathsf{b}}^{\mathsf{w}}(\mathsf{i})$ such that the party Q_i contributes $\mathsf{k}_{\mathsf{b}}^{\mathsf{w}}(\mathsf{i})$. Wire w will also have a mask $\lambda^w = \lambda_1^w \oplus \ldots \oplus \lambda_n^w$ such that λ_i^w is contributed by party Q_i. Now, the transformed C will have input wires corresponding to each bit of $\mathsf{key}_{\mathsf{0}}^{\mathsf{w}}$ and $\mathsf{key}_{\mathsf{1}}^{\mathsf{w}}$ and also λ^w. That is, the circuits along with having keys of some other circuits, will also have masks for the corresponding wires.[14] This is necessary for consistency and correctness of evaluation of the garbled circuits. We further expand the input λ^w into n bits as $\lambda_1^w, \ldots, \lambda_n^w$. Finally, input wires of C corresponding to $\mathsf{k}_{\mathsf{b}}^{\mathsf{w}}(\mathsf{i})$ for $b \in \{0, 1\}$ and λ_i^w will correspond to input wires of party Q_i. Note that the transformed circuit falls in the framework of [2].

[13] This would be useful in arguing that the functionality computed under generic MPC is information theoretic as well as arguing efficiency while garbling multiple programs w.r.t. a persistent database.

[14] The circuits described in [11] will be modified naturally to include these mask bits in evaluation. For example, consider a circuit which was originally producing output qKey_x, where qKey was a collection of keys (two per bit) being selected by string x. Now new circuit will output $\mathsf{qKey}_{x \oplus \boldsymbol{\lambda}}$, where $\boldsymbol{\lambda}$ contains the mask bits for all wires in x. Hence, if qKey was given as input, then we also need to provide $\boldsymbol{\lambda}$ as input.

5.3 Semi-Honest Secure Protocol for RAM Computation

In this section, we describe our constant-round protocol for semi-honest secure multiparty RAM computation that only makes a black-box use of one-way functions in the OT-hybrid model. The parties will run the semi-honest protocol from [2] to collectively generate the garbled RAM and compute the garbled RAM locally to obtain the output. As mentioned before, a naïve implementation of this idea results in a non-black-box use of one-way functions because a garbled RAM consists of a bunch of garbled circuits (that use PRG inside them). To overcome this issue, we will use the garbled RAM instantiation based on garbling scheme of [2] described above (see Lemma 3) without the hard-coding of keys. Here, the main idea is that each party invokes the one-way function locally and off-the-shelf secure computation protocol is invoked only for an information theoretic functionality (that we describe below). Moreover, recall that the garbled RAM scheme with full security compiles a UMA-2 secure scheme with a statistically secure ORAM. It is crucial for black-box nature of our protocol that the ORAM scheme is statistical and does not use any cryptographic primitives. Hence, intuitively, since the secure computation protocol of [2] is black-box, the overall transformation is black-box in use of OWFs.

The functionality $\mathcal{F}_{\mathbf{Gram}}$. We begin by describing the functionality $\mathcal{F}_{\text{GRAM}}$ w.r.t. a program P that will be computed by the parties via the constant-round black-box protocol of [2].

1. **Inputs:** The input of party Q_i consists of x_i, database D_i, shares of keys for all the garbled circuits denoted by Labels_i, all the wire masks denoted by λ_i, the relevant PRG outputs on labels denoted by PRG_i, and randomness for ORAM r_i.
2. **Output:** Each party gets the garbled RAM $(\tilde{D}, \tilde{P}, \tilde{x})$ for program P promised by Lemma 3. The randomness used by ORAM is computed as $\oplus_{i \in [n]} r_i$.

Now we argue that the above described functionality is *information theoretic*. We first note that garbled RAM consists of a collection of garbled circuits and the structure of these circuits as well as interconnection of these circuits is known publicly. This is true because we have removed all the sensitive information that was earlier hard-coded as additional input to the circuits. Moreover, the circuits that are being garbled in [11] are information theoretic. This follows from the fact that [11] only makes a black-box of one-way functions. Secondly, once all the labels as well as PRG outputs are computed locally by the parties, the garble table generation is information theoretic (see Section 5.1). Thirdly, the circuit to compute the labels for partial inputs that are revealed at the time of garbled RAM generation is information theoretic. This is because the values of those partial inputs can be computed information theoretically from the inputs of the parties. And finally, we use the fact that the ORAM used is statistical and hence, information theoretic. Therefore, the overall functionality $\mathcal{F}_{\text{GRAM}}$ is information theoretic.

Our protocol. Consider n parties $Q_1, \ldots, Q_n$ who want to compute a program P. The party Q_i holds an input x_i and database D_i. Let us denote the semi-honest protocol for RAM computation of P by $\Pi^{\mathsf{sh}}_{\mathsf{RAM}}$ that is as follows:

- **Step 1.** Party Q_i computes the inputs for the functionality $\mathcal{F}_{\text{GRAM}}$ described above. More precisely, party Q_i does the following: It picks a seed s_i for a PRF and randomness r_i for ORAM. It generates its shares for keys to all wires of all the circuits Labels_i by computing the PRF with key s_i on appropriate inputs. It also picks a random mask for each wire λ_i. Party Q_i also locally computes the relevant PRG outputs PRG_i needed for garbled tables (see Section 5.1).
- **Step 2.** The parties run the semi-honest secure protocol $\Pi^{\mathcal{F}_{\text{GRAM}}}_{\mathsf{bmr}}$ (provided by Theorem 3) to compute the functionality $\mathcal{F}_{\text{GRAM}}$ described above. Each party will get the garbled RAM $(\tilde{D}, \tilde{P}, \tilde{x})$ as output.
- **Step 3.** Each party runs $\mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x})$ to obtain the output y.

Correctness. The correctness of the above protocol follows trivially from the correctness of $\Pi^{\mathcal{F}_{\text{GRAM}}}_{\mathsf{bmr}}$ and correctness of garbled RAM of [11].

Round complexity. The round complexity of the above protocol is same as the round complexity of $\Pi^{\mathcal{F}_{\text{GRAM}}}_{\mathsf{bmr}}$. Hence, it is a constant by Theorem 3.

Black-box use of one-way functions. This follows from the fact that the protocol $\Pi^{\mathcal{F}}_{\mathsf{bmr}}$ in Theorem 3 only makes a black-box use of one-way functions in the OT-hybrid model and that the functionality $\mathcal{F}_{\text{GRAM}}$ is set up to be information theoretic (as argued above).

Efficiency. First of all, [11] guarantees that the number of circuits needed for $\tilde{D}$ is only proportional to $|D|$ up to poly-logarithmic factors, and that the number of circuits needed for $\tilde{P}$ is proportional to the running time of P up to poly-logarithmic factors. The functionality $\mathcal{F}_{\text{GRAM}}$ that generates the garbled RAM ($\tilde{D}$ and $\tilde{P}$) has size linear in the garbled RAM itself. And finally, the communication and computation complexities of $\Pi^{\mathcal{F}_{\text{GRAM}}}_{\mathsf{bmr}}$ grow linearly in the size of $\mathcal{F}_{\text{GRAM}}$. Therefore the entire communication and computation complexities are satisfactory.

Note that for the desired efficiency, it is crucial that we run a single MPC protocol to generate all the garbled circuits instead of running multiple sessions of MPC (each generating one garbled circuit) in parallel. This is because partial labels for some garbled circuits depend on the actual database values and for security it is crucial to hide which circuit corresponds to which index in the database. This security guarantee is achieved by the use of ORAM in the garbled RAM scheme. In our protocol, for security, the randomness of ORAM is contributed by all the parties. Hence, separating the generation of garbled circuits would require all the parties to input the entire database to each garbled circuit generation, and the resulting efficiency of garbling the database would be at least quadratic in its size.

5.4 Proof of Semi-Honest Security

In this section, we prove that the above protocol is secure against a semi-honest adversary corrupting a set $\mathcal{I} \subseteq [n]$ of parties. We would rely on the semi-honest security of garbled RAM from [11] when instantiated using the BMR garbling scheme (see Lemma 3) as well as semi-honest security of $\Pi_{\mathsf{bmr}}^{\mathcal{F}}$ (see Theorem 3). We will define our simulator $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ and prove that the view computed by $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ is indistinguishable from the view of the adversary in the real world.[15]

1. $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ begins by corrupting the parties in set $\mathcal{I}$ and obtains the inputs $x_{\mathcal{I}} = \{x_i\}_{i \in \mathcal{I}}$ and database $D_{\mathcal{I}} = \{D_i\}_{i \in \mathcal{I}}$ of the corrupt parties. $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ queries trusted functionality for program P on input $(x_{\mathcal{I}}, D_{\mathcal{I}})$ and receives output y.
2. $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ picks PRF keys s_i and randomness r_i for ORAM for all $i \in \mathcal{I}$. It computes the shares of keys for all the wires of all the circuits Labels_i, wire masks λ_i as well as PRG outputs PRG_i honestly for all the corrupt parties.
3. $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ invokes the simulator of the garbled RAM $\mathsf{GramSim}_{\mathsf{bmr}}(1^\kappa, 1^t, 1^{|D|}$, $(x_{\mathcal{I}}, D_{\mathcal{I}}, \mathsf{Labels}_{\mathcal{I}}), y)$ to get the simulated garbled RAM. Let us denote it by $(\tilde{D}, \tilde{P}, \tilde{x})$.
4. $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ now invokes $\mathsf{Sim}_{\mathsf{bmr}}$ for functionality $\mathcal{F}_{\text{GRAM}}$ where the inputs of the corrupt parties are obtained in Step 2 above, that is, $\{(x_i, D_i, r_i, \mathsf{Labels}_i, \lambda_i, \mathsf{PRG}_i)\}_{i \in \mathcal{I}}$ and output is the simulated garbled RAM $(\tilde{D}, \tilde{P}, \tilde{x})$. More precisely, the simulator $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ outputs $\mathsf{Sim}_{\mathsf{bmr}}(1^\kappa, \{(x_i, D_i, r_i, \mathsf{Labels}_i, \lambda_i, \mathsf{PRG}_i)\}_{i \in \mathcal{I}}, (\tilde{D}, \tilde{P}, \tilde{x}))$.

Next, we show that the output of the simulator $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ is indistinguishable from the real execution via a sequence of hybrids $\mathsf{Hyb}_0, \mathsf{Hyb}_1, \mathsf{Hyb}_2$, where we prove that the output of any pair of consecutive hybrids is indistinguishable. Hyb_0 corresponds to the real execution and Hyb_2 corresponds to the final simulation.

Hyb_1*:* This is same as the hybrid Hyb_0 except we simulate the protocol for $\Pi_{\mathsf{bmr}}^{\mathcal{F}_{\text{GRAM}}}$ by using $\mathsf{Sim}_{\mathsf{bmr}}$ with real garbled RAM as output. More precisely, this hybrid is as follows:

1. For all $i \in [n]$, let x_i and D_i denote the input and database respectively.
2. Pick PRF keys s_i and randomness r_i for ORAM for all $i \in [n]$.
3. For each $i \in \mathcal{I}$, compute the shares of keys for all the wires of all the circuits, wire masks as well as PRG outputs honestly for the corrupt parties. For the party Q_i, denote the collection of wire key shares by Labels_i, collection of wire masks shares by λ_i and the collection of PRG outputs by PRG_i.
4. Generate the garbled RAM $(\tilde{D}, \tilde{P}, \tilde{x})$ honestly as $\mathsf{Gram}_{\mathsf{bmr}}(1^\kappa, 1^t, D_{[n]}, P, x_{[n]}, s_{[n]})$.
5. Run $\mathsf{Sim}_{\mathsf{bmr}}$ for functionality $\mathcal{F}_{\text{GRAM}}$ where the inputs of the corrupt parties are obtained in Steps 2 and 3 above, that is, $\{(x_i, D_i, r_i, \mathsf{Labels}_i, \lambda_i, \mathsf{PRG}_i)\}_{i \in \mathcal{I}}$ and output is the garbled RAM $(\tilde{D}, \tilde{P}, \tilde{x})$ obtained above. More precisely, the simulator $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{sh}}$ outputs $\mathsf{Sim}_{\mathsf{bmr}}(1^\kappa, \{(x_i, D_i, r_i, \mathsf{Labels}_i, \lambda_i, \mathsf{PRG}_i)\}_{i \in \mathcal{I}}, (\tilde{D}, \tilde{P}, \tilde{x}))$.

[15] In the semi-honest setting, outputs of the honest parties are identical in the real world and the ideal world.

Intuitively, $\mathsf{Hyb}_0 \approx_c \mathsf{Hyb}_1$ by the correctness and the semi-honest security of the multi-party protocol from [2] used for computing the garbled RAM (see Theorem 3 for formal security guarantee).

Hyb_2*:* This is same as simulator $\mathsf{Sim}^{\mathsf{sh}}_{\mathsf{mpc}}$. In other words, this is same as the previous hybrid except instead of using actual circuits of garbled RAM, we use simulated garbled circuits output by $\mathsf{GramSim}_{\mathsf{bmr}}$ on $\mathsf{GramSim}_{\mathsf{bmr}}(1^\kappa, 1^t, 1^{|D|}$, $(x_{\mathcal{I}}, D_{\mathcal{I}}, \mathsf{Labels}_{\mathcal{I}}), y)$. Note that unlike previous hybrid, this hybrid does not rely on the inputs and the database of honest parties.

Hybrids $\mathsf{Hyb}_1 \approx_c \mathsf{Hyb}_2$ by Lemma 3.

5.5 Running More Than One Program On A Persistent Database

In this section, we provide a protocol for executing multiple programs $P^{(1)}, \ldots, P^{(\ell)}$ on a persistent database. For exposition, it suffices to describe the case of $P^{(1)}$ and $P^{(2)}$, and it would be easy to extend to more programs in a natural way.

Recall that the garbled memory consists of a collection of garbled circuits that get consumed when a program is executed. Note that for security each garbled circuit can only be executed on a single input. Hence, the main issue in executing multiple programs on a persistent memory (as already observed by [11]) is that we need a way to replenish the circuits in the memory so that we can allow for more reads/writes by programs. To address this challenge, [11] gave a mechanism for replenishing circuits obliviously where each garbled program running for time T also generates enough garbled circuits for memory to support T more reads. Recall that the garbled memory consists of a tree of garbled circuits where each node consists of a sequence of garbled circuits. In [11], since the execution of the program $P^{(1)}$ as well as which circuits get consumed is hidden from the garbler, they need a more sophisticated oblivious technique for replenishing. This can be simplified in our setting because all the parties execute the garbled RAM for $D, P^{(1)}$ and hence, know which garbled circuits have been consumed and need to be replenished at any node of the tree.

At the time of garbling of the second program $P^{(2)}$, the parties would compute the functionality $\mathcal{F}_{\text{REP}}$ described next.

The functionality $\mathcal{F}_{\mathbf{Rep}}$. This functionality guarantees that the parties replenish the garbled memory to support M reads before executing the program $P^{(2)}$, where M is the size of the database. The garbled circuits that need to be replenished is determined by the execution of $P^{(1)}$ and hence, is known to all the parties.

1. **Inputs:** The input of party Q_i consists of x_i (inputs for program $P^{(2)}$), shares of keys, wire masks needed for the garbled circuits of program $P^{(2)}$ as well as share of keys, wire masks for the memory garbled circuits to be added. Note that these keys and wire masks for garbled circuits of the program and memory need to be consistent with previously generated garbled circuits. As before, we denote these by Labels_i and λ_i respectively. Parties also generate the relevant PRG outputs on labels denoted by PRG_i.

2. **Output:** Each party gets the garbled program and garble input $(\tilde{P}^{(2)}, \tilde{x}^{(2)})$ for program $P^{(2)}$, as well as more garbled circuits for the memory which restores the garbled memory to support M more reads.

Our protocol. We describe the protocol for running $P^{(1)}$ and $P^{(2)}$ on a persistent database D below. Note that at a high level, the following invariant is maintained. Before executing a program, the memory is replenished to support M reads, where M is the size of the memory (which is w.l.o.g. greater than the running time of the program).

- **Step 1.** The parties $Q_1, \ldots, Q_n$ run the protocol $\Pi^{\mathsf{sh}}_{\mathsf{RAM}}$ to generate the garbled RAM $\tilde{D}^{(1)}, \tilde{P}^{(1)}, \tilde{x}^{(1)}$. Each party Q_i executes the above garbled RAM to obtain the output $y^{(1)}$ and the resulting garbled memory $\tilde{D'}^{(1)}$. (The parties know what all garbled circuits got consumed from the garbled memory and need to be replenished.)
- **Step 2.** Party Q_i computes the inputs for the functionality $\mathcal{F}_{\textsc{Rep}}$ described above. More precisely, party Q_i does the following: It uses its PRF seed s_i to generate its shares for all keys needed for generating the new circuits Labels_i. It picks a random mask for each wire λ_i, and locally computes the relevant PRG outputs PRG_i needed for garbled tables. Note that the shares for keys and wire masks need to be consistent with previously generated garbled circuits. Since we are in the semi-honest setting, we can safely assume that the consistency is maintained.
- **Step 3.** The parties run the semi-honest secure protocol $\Pi^{\mathcal{F}_{\textsc{Rep}}}_{\mathsf{bmr}}$ to compute the functionality $\mathcal{F}_{\textsc{Rep}}$ described above. Each party will obtain the garbled program and garbled input $(\tilde{P}^{(2)}, \tilde{x}^{(2)})$. Each party will also get more circuits for the memory which restores the garbled memory to support M more reads, thus obtaining an updated garbled memory $\tilde{D}^{(2)}$.
- **Step 4.** Each party runs $\mathsf{GEval}^{\tilde{D}^{(2)}}(\tilde{P}^{(2)}, \tilde{x}^{(2)})$ to obtain the output $y^{(2)}$ and modified garbled memory $\tilde{D'}^{(2)}$.

The *correctness*, *constant round complexity*, and *black-box use of one-way functions* of the protocol can be argued similarly as before in Section 5.4. A crucial idea in arguing correctness is that the structure of the garbled circuits needed is public and the keys of new circuits should be consistent with previously generated garbled circuits. This is easy to ensure as the parties behave honestly in generating consistent labels using the same PRF keys as before.

Efficiency. We argue that the complexity of garbling second program is proportional to the running time of $P^{(2)}$ up to poly-logarithmic factors. The argument follows in a similar manner as Section 5.4. First note that the number of garbled circuits in garbled program $\tilde{P}^{(2)}$ is proportional to the running time of $P^{(2)}$. The replenishing mechanism of [11] ensures that the number of new circuits needed is only proportional to the running time of $P^{(1)}$ up to poly-logarithmic factors (since these are the number of circuits consumed from memory while running $P^{(1)}$ program).[16]

It is crucial to note that the cost of replenishing does not grow with the size of the memory. We note that replenishing does not require the database contents or the randomness used by ORAM as input. More precisely, for any node in the tree of the garbled memory, adding more circuits to that node only requires knowledge of the labels used in the previous circuit in that node and being consistent with that.[17]

Security. We can argue the semi-honest security of the above protocol as follows: The simulator needs to simulate the view of the adversary for all the program executions using the corresponding simulator of garbled RAM provided by [11] as before. The only non-triviality is that the underlying simulator of garbled RAM would need to know the outputs of all the executions in order to do successful simulation of garbled RAM. Since we are the semi-honest setting, these are easy to compute. This is because since all the parties are semi-honest, the parties choose the inputs for different execution just based on their previous inputs and outputs. This choice does not depend on the garbled RAM given to the parties in the real execution. This will be the major bottleneck for the malicious setting and we will revisit this point later.[18]

At a high level, our simulator for the persistent database would do the following. As before, let $\mathcal{I} \subset [n]$ denote the set of corrupt parties. The simulator will use $D_{\mathcal{I}}$ and $x_{\mathcal{I}}^{(1)}$ of corrupt parties to learn the output $y^{(1)}$ from the trusted functionality. Then, it will use the honest party strategy to compute the next round of adversarial inputs $x_{\mathcal{I}}^{(2)}$ for the second program, and learn the output $y^{(2)}$. This way the simulator learns the outputs $y^{(1)}, \ldots, y^{(\ell)}$ for all the executions. Now we can use the simulator of [11] to simulate the garbled RAM consisting of all the garbled circuits. Finally, we use the simulator $\mathsf{Sim}_{\mathsf{bmr}}$ on garbled RAM to simulate the view of adversary for all the program executions. The argument of indistinguishability follows in the same manner as proof of a single program in Section 5.4.

5.6 Load Balancing

In the protocol we have described above for semi-honest RAM computation, each party gets a garbled RAM, which can be computed locally. This requires each party to store information whose size is at least as large as size of total database. In some settings, this might not be desired. In the semi-honest setting it is easy to guarantee security even when only one party stores the garbled memory and

[16] For ease of calculation, we can include of replenishing after running $P^{(1)}$ in the cost of $P^{(1)}$ and cost of replenishing after $P^{(2)}$ in the cost of $P^{(2)}$ and so on.

[17] This is because the keys of the successor circuit are already fed as input to the predecessor and the new circuit just has to be consistent to the previously generated circuit.

[18] Note that a malicious party might choose its inputs for the second program execution based on the garbled memory obtained in the first execution.

computes the program. We can simply modify the protocol such that only one party gets the garbled RAM as output. All the other parties get no output. The party which gets the garbled RAM, computes it, and sends the output of the computation to all the other parties. Since we are in the semi-honest setting, it is easy to see that this protocol is correct and secure given the correctness and security of the original protocol.

6 Malicious Setting

In this section, we show how the semi-honest protocol presented in Section 5 can be extended using appropriate tools to work for the malicious setting in the case of non-persistent database.[19] We show the following result:

Theorem 4. *There exists a constant-round malicious secure multiparty protocol in the OT-hybrid model that makes a black-box use of one-way functions for secure RAM computation for the case of non-persistent database satisfying security and efficiency requirements of Section 4.*

Protocol Overview. We first recall the semi-honest protocol from Section 5.3 at a high level. In the first step, each party picks a seed of a PRF and computes the share of keys and masks for all the circuits in the garbled RAM as output of the PRF as well as the PRG outputs on all the key shares. The parties also pick randomness for statistical ORAM such that final randomness used is the sum of the randomness from all the parties. Next, the parties run the constant round protocol of [2] to generate all the garbled circuits for the garbled RAM of [11] instantiated with distributed garbling scheme (see Lemma 3). A key point was that the functionality $\mathcal{F}_{\text{GRAM}}$ executed via the secure computation protocol is information theoretic that gives us a black-box constant-round protocol. In this section, we would extend these ideas to construct a malicious secure protocol as follows:

To protect against malicious behavior, we need to ensure that the adversary behaves honestly in the above high-level protocol. Towards this, we transform the above protocol as follows: The first step is to replace the execution of $\Pi_{\mathsf{bmr}}^{\mathcal{F}_{\text{GRAM}}}$ with a malicious secure computation protocol. Since the overall goal is to get a constant round protocol for RAM which makes a black-box use of PRG in the OT-hybrid model, we use the protocol from [30] that gives such a protocol for circuits. Denote this protocol by $\Pi_{\mathsf{ips}}^{\mathcal{F}_{\text{GRAM}}}$. More formally, the following theorem was proven by [30].

Theorem 5 ([30], Theorem 3). *For any $n \geq 2$ there exists an n-party constant-round secure computation protocol in the OT-hybrid model which makes a black-box use of a pseudorandom generator and achieves computational UC-security against an active adversary which may adaptively corrupt at most $n-1$ parties.*

[19] We address the issue of persistent database at the end of this section.

Let $\mathsf{Sim}_{\mathsf{ips}}$ be the simulator provided by the above theorem. In our case, we only need stand-alone security against a static malicious adversary which is weaker than what is provided by the above theorem.

Next, recall that in a distributed garbling scheme of Section 5.2, each party computes shares of labels using a PRF seed as well as PRG outputs on these labels to be used in garbling of individual gates. If we can ensure that a malicious party correctly computes the labels as outputs of a PRF and also the PRG outputs, intuitively the security would follow similarly to the semi-honest scenario by relying on the malicious security of $\Pi_{\mathsf{ips}}^{\mathcal{F}_{\text{GRAM}}}$. Since the goal is to construct a protocol that only makes a black-box of one-way functions, we cannot make the parties prove that they computed the PRF or PRG values correctly. In particular, proving correctness of PRF and PRG outputs would lead to a non-black-box use of cryptography. To solve this issue we make two key observations described below.

First, we observe that in any scheme for circuit garbling as well as garbled RAM, the wire labels are chosen to be outputs of a PRF only for efficiency and not security. That is, even if malicious parties choose these labels as arbitrarily chosen strings, it does not compromise the security of the garbled circuits, or garbled RAM scheme, and hence, our construction. But, the correct computation of PRG values used to encrypt the keys in garbled gate tables, is indeed critical. In fact, if a malicious party feeds the wrong output of PRG in computing of the garbled tables, it can cause the honest parties to abort during evaluation of garbled circuits or garbled RAM.[20] Note that an adversary can choose to cause this abort selectively by computing, let us say, most PRG outputs correctly and only a few incorrectly. This seems highly problematic at first, since this can lead to the problem of selective abort based on the inputs of honest parties and would break security. Our key observation is that the probability of this abort happening is independent of the inputs of the honest parties and is, indeed, simulatable given just the labels and the PRG outputs used by the adversary during the secure computation. This holds because of the following:

1. In the distributed garbling scheme, during evaluation, all parties decrypt the same row during evaluation (See Sections 5.1, 5.2 for details). That is, the adversary as well as the honest parties decrypt the same row for all the gate tables. Now, since this scheme is semi-honest secure, which row is decrypted during evaluation is independent of the honest parties' inputs. In fact, it depends on the mask value λ^w for the wires which is secret shared among all parties and hence, hidden from all the parties.
2. The protocol $\Pi_{\mathsf{ips}}^{\mathcal{F}_{\text{GRAM}}}$ is a correct and malicious secure protocol. Hence, the simulator of $\Pi_{\mathsf{ips}}^{\mathcal{F}_{\text{GRAM}}}$ would extract an input for the adversary that consists of $x_i, D_i, r_i, \mathsf{Labels}_i, \mathsf{PRG}_i$ for all corrupt parties. Here, Labels_i and PRG_i correspond to the label shares and PRG outputs used by Q_i. Looking at

[20] Recall that during evaluation, when a party Q_i decrypts the key for the output of a gate, it aborts if the i^{th} sub-part of the key does not match either the 0 or 1 key of Q_i.

these, the simulator can check which PRG outputs have been computed incorrectly. Moreover, each PRG value is used in exactly one row of one gate table that is fixed. Now, as mentioned before, incorrect PRG values can cause an honest party to abort. But, whether Q_i aborts or not is independent of input of Q_i or any other party because of the following: The adversary feeds a PRG value to mask the keys in this row of garbled table, which acts independently on $\mathsf{k}_\mathsf{b}^\mathsf{w}(\mathsf{i})$ for each i. This is because $\mathsf{key}_\mathsf{b}^\mathsf{w} = \mathsf{k}_\mathsf{b}^\mathsf{w}(1) \circ \ldots \circ \mathsf{k}_\mathsf{b}^\mathsf{w}(\mathsf{n})$. If the PRG value used by the adversary does not match the correct PRG output for masking $\mathsf{k}_\mathsf{b}^\mathsf{w}(\mathsf{i})$, then w.h.p. it would not match the keys of Q_i for both 0 and 1 and the party Q_i would abort. This behavior is completely simulatable just given the input of the adversary.

In short, the adversary can only control whether an honest party Q_i aborts or not on some specific gate. The adversary cannot set up the incorrect PRG values to change the label from 0 to 1 for an honest party because honest labels are chosen to be outputs of a PRF. We can continue the same argument for each gate to conclude that the adversary cannot make an honest party compute a wrong output.

Recall that each garbled gate has 4 rows of encryptions. For any gate, the adversary can behave honestly for α rows of the gate and cheat in $4 - \alpha$ rows. In this case, the honest party would abort with probability $1 - \alpha/4$, again independent of inputs as which row gets decrypted during evaluation is uniform (depends only on mask values). This cheating behavior is simulatable as well.

6.1 Our Protocol

We now describe our protocol for constant-round malicious secure RAM computation denoted by $\Pi_{\mathsf{RAM}}^{\mathsf{mal}}$. This protocol is same as the semi-honest secure protocol with one change that we use malicious secure protocol $\Pi_{\mathsf{ips}}^{\mathcal{F}_{\text{GRAM}}}$ from [30] instead of semi-honest secure $\Pi_{\mathsf{bmr}}^{\mathcal{F}_{\text{GRAM}}}$ to compute the garbled RAM in a distributed manner.

Recall the functionality $\mathcal{F}_{\text{GRAM}}$ described in Section 5.3 that takes as input the shares of randomness for ORAM, keys for all wires of all the garbled circuits, PRG outputs used in generating garbled tables (see Section 5.1), share of wire masks as well as inputs $x_1, \ldots, x_n$ and data base $D_1, \ldots, D_n$ and produces the corresponding garbled RAM for P as output. The randomness used for ORAM is the sum of the shares of randomness from all the parties. Note that as argued in Section 5.3, the functionality $\mathcal{F}_{\text{GRAM}}$ is information theoretic and does not use oneway function or any other cryptographic primitives.[21] Now the protocol $\Pi_{\mathsf{RAM}}^{\mathsf{mal}}$ is as follows:

- **Step 1.** Party Q_i computes the inputs for the functionality $\mathcal{F}_{\text{GRAM}}$. More precisely, party Q_i does the following: It picks a seed s_i for a PRF and randomness r_i for ORAM. It generates its shares for keys to all wires of all

[21] This is because ORAM is statistical and garble table generation is information theoretic once wire labels as well PRG outputs are known.

the circuits Labels_i by computing the PRF with key s_i on appropriate inputs. It also picks a random mask for each wire λ_i. Party Q_i also locally computes the relevant PRG outputs PRG_i needed for garbled tables (see Section 5.1).

- **Step 2.** The parties run the constant round malicious secure protocol $\Pi_{\mathsf{ips}}^{\mathcal{F}_{\textsc{Gram}}}$ (provided by [30]) to compute the functionality $\mathcal{F}_{\textsc{Gram}}$. Each party will get the garbled RAM $(\tilde{D}, \tilde{P}, \tilde{x})$ as output.
- **Step 3.** Each party runs $\mathsf{GEval}^{\tilde{D}}(\tilde{P}, \tilde{x})$ to obtain the output[22] y.

Similar to the semi-honest case, the correctness, the constant round-complexity and the black-box nature of the above protocol follow in a straightforward manner from the corresponding properties of $\Pi_{\mathsf{ips}}^{\mathcal{F}_{\textsc{Gram}}}$ as well as garbled RAM of [11].

The formal proof of malicious security of this protocol appears in Section 6.2.

The case of persistent database. As noted earlier, we do not handle the case of persistent database against malicious adversaries, as remarked below:

Remark 3. We leave open the problem of realizing a solution against malicious adversaries for the persistent database setting. Realizing a solution that supports persistent database would involve realizing garbled RAM with *adaptive input security*. Specifically, a garbled RAM solution for which the inputs on which the persistent garbled RAM is invoked can be chosen depending on the provided garbled RAM itself. In order to construct such a scheme, efficient garbled circuit construction satisfying analogous stronger security properties is needed. No construction for such garbled circuits are known based on the standard assumptions (i.e., without random-oracle model).

6.2 Proof of Malicious Security

In this section, we prove that the above protocol is secure against a malicious adversary corrupting a set $\mathcal{I} \subseteq [n]$ of parties. We will construct a simulator $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{mal}}$ and prove that the joint distribution of the view computed by $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{mal}}$ and the outputs of the honest parties is indistinguishable from the real world.

Let us denote the collection of labels used by party Q_i by Labels_i, set of PRG outputs by PRG_i and the set of wire masks by λ_i. Note that while generating the garbled RAM, an honest party uses the correct PRG outputs but a malicious party may use arbitrary strings. Let $\widehat{\mathsf{PRG}}_i$ denote the correct PRG outputs corresponding to Labels_i. $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{mal}}$ works as follows.

1. $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{mal}}$ begins by corrupting the parties in set $\mathcal{I}$.
2. Next, $\mathsf{Sim}_{\mathsf{mpc}}^{\mathsf{mal}}$ runs $\mathsf{Sim}_{\mathsf{ips}}$ for functionality $\mathcal{F}_{\textsc{Gram}}$ that would begin by extracting the inputs $x_{\mathcal{I}} = \{x_i\}_{i \in \mathcal{I}}$ and database $D_{\mathcal{I}} = \{D_i\}_{i \in \mathcal{I}}$ of the corrupt parties as well as the collection of labels $\mathsf{Labels}_{\mathcal{I}}$, PRG values $\mathsf{PRG}_{\mathcal{I}}$, wire masks $\lambda_{\mathcal{I}}$ and ORAM randomness share $r_{\mathcal{I}}$.

[22] Since we are in the malicious setting, some honest parties may output $\perp$. This would be captured by the ideal world adversary described later.

3. $\mathsf{Sim}^{\mathsf{mal}}_{\mathsf{mpc}}$ queries trusted functionality for program P on input $(x_{\mathcal{I}}, D_{\mathcal{I}})$ and receives output y.
4. $\mathsf{Sim}^{\mathsf{mal}}_{\mathsf{mpc}}$ runs a simulator $\mathsf{GramSim}'_{\mathsf{bmr}}$ on $(1^{\kappa}, 1^{t}, 1^{|D|}, (x_{\mathcal{I}}, D_{\mathcal{I}}, \mathsf{Labels}_{\mathcal{I}}, \mathsf{PRG}_{\mathcal{I}}), y)$ to obtain the simulated garbled RAM. Let us denote it by $(\tilde{D}, \tilde{P}, \tilde{x})$. Here, $\mathsf{GramSim}'_{\mathsf{bmr}}$ denotes the stronger version[23] of $\mathsf{GramSim}_{\mathsf{bmr}}$ that uses $\mathsf{PRG}_{\mathcal{I}}$ instead of $\widehat{\mathsf{PRG}}_{\mathcal{I}}$. We describe this simulator formally in full version.
5. $\mathsf{Sim}^{\mathsf{mal}}_{\mathsf{mpc}}$ gives $(\tilde{D}, \tilde{P}, \tilde{x})$ to $\mathsf{Sim}_{\mathsf{ips}}$ as the output of the secure computation protocol of $\Pi^{\mathcal{F}_{\text{GRAM}}}_{\mathsf{ips}}$. $\mathsf{Sim}_{\mathsf{ips}}$ will now simulate the view of the adversary in the protocol $\Pi^{\mathcal{F}_{\text{GRAM}}}_{\mathsf{ips}}$.
6. $\mathsf{Sim}^{\mathsf{mal}}_{\mathsf{mpc}}$ computes the set $\mathcal{J} \subseteq [n] \backslash \mathcal{I}$ of honest parties who receive the output. Details on how to do this are described below. $\mathsf{Sim}^{\mathsf{mal}}_{\mathsf{mpc}}$ will now send $\mathcal{J}$ to the trusted functionality.

Computing the set $\mathcal{J}$ of honest parties who receive the output in ideal world. $\mathsf{Sim}^{\mathsf{mal}}_{\mathsf{mpc}}$ runs $\mathsf{GramSim}_{\mathsf{bmr}}((1^{\kappa}, 1^{t}, 1^{|D|}, (x_{\mathcal{I}}, D_{\mathcal{I}}, \mathsf{Labels}_{\mathcal{I}}), y)$ to compute the honest garbled RAM $(\hat{D}, \hat{P}, \hat{x})$ and executes it. This defines the set of relevant rows as the ones that need to be decrypted in any gate that is executed for any garbled circuit. Note that an honest party aborts iff decryption of at least one of the relevant row fails. Moreover, decryption fails for party Q_j if the j^{th} part of the decrypted label matches neither the party's label 0 nor label 1.

For simplicity, consider one such relevant row that is used during execution. W.l.o.g. it is enough to consider the xor of PRG values input by the adversary on behalf of all corrupt parties. For this row, let $\hat{\mathsf{prg}} = \hat{a}_1 \circ \ldots \circ \hat{a}_n$ define the xor of correct PRG values from the adversary and $\mathsf{prg} = a_1 \circ \ldots \circ a_n$ denote the xor of PRG values used by the adversary. Here, intuitively, $\hat{a}_j$ or a_j defines the part of PRG value used to mask the part of the label contributed by party Q_j. See Section 5.1 for details of each gate garbling. The index $j \in [n] \setminus \mathcal{I}$ belongs to set $\mathcal{J}$ iff $\hat{a}_j = a_j$ for all relevant rows. That is, only then the honest party Q_j is able to decrypt all the relevant rows correctly. Note that since honest party chooses its labels as outputs of a PRF, the encryption of label 0 cannot be decrypted as label 1. This happens only when $\hat{a}_j \oplus a_j = \mathsf{k}^{\mathsf{w}}_0(\mathsf{j}) \oplus \mathsf{k}^{\mathsf{w}}_1(\mathsf{j})$, which happens with negligible probability in κ.

For a formal proof of indistinguishability of views of real and ideal worlds, refer to the full version of the paper.

References

1. Beaver, D.: Correlated pseudorandomness and the complexity of private computations. In: 28th ACM STOC. pp. 479–488. ACM Press (May 1996)

[23] Recall that PRG outputs are only used to mask the keys of the output wire (see gate garbled technique in Section 5.1). This simulator uses the PRG values provided by the adversary instead of correct PRG outputs. This change can be described by a simple deterministic transformation on output of $\mathsf{GramSim}_{\mathsf{bmr}}$.

2. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols (extended abstract). In: 22nd ACM STOC. pp. 503–513. ACM Press (May 1990)
3. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 12. pp. 784–796. ACM Press (Oct 2012)
4. Bitansky, N., Garg, S., Telang, S.: Succinct randomized encodings and their applications. Cryptology ePrint Archive, Report 2014/771 (2014), http://eprint.iacr.org/2014/771
5. Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications. In: STOC. pp. 103–112 (1988)
6. Canetti, R., Holmgren, J., Jain, A., Vaikuntanathan, V.: Indistinguishability obfuscation of iterated circuits and RAM programs. Cryptology ePrint Archive, Report 2014/769 (2014), http://eprint.iacr.org/2014/769
7. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. J. Comput. Syst. Sci. 7(4), 354–375 (1973)
8. Feige, U., Lapidot, D., Shamir, A.: Multiple non-interactive zero knowledge proofs under general assumptions. SIAM Journal of Computing 29(1), 1–28 (1999)
9. Garg, S., Gentry, C., Halevi, S.: Candidate multilinear maps from ideal lattices. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 1–17. Springer, Heidelberg (May 2013)
10. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: 54th FOCS. pp. 40–49. IEEE Computer Society Press (Oct 2013)
11. Garg, S., Lu, S., Ostrovsky, R.: Black-box garbled RAM. In: 56th Annual IEEE Symposium on Foundations of Computer Science (2015)
12. Garg, S., Lu, S., Ostrovsky, R., Scafuro, A.: Garbled RAM from one-way functions. In: Servedio, R.A., Rubinfeld, R. (eds.) 47th ACM STOC. pp. 449–458. ACM Press (Jun 2015)
13. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) 41st ACM STOC. pp. 169–178. ACM Press (May / Jun 2009)
14. Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., Wichs, D.: Garbled RAM revisited. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 405–422. Springer, Heidelberg (May 2014)
15. Gentry, C., Halevi, S., Raykova, M., Wichs, D.: Outsourcing private RAM computation. In: 55th FOCS. pp. 404–413. IEEE Computer Society Press (Oct 2014)
16. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: Aho, A. (ed.) 19th ACM STOC. pp. 182–194. ACM Press (May 1987)
17. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC. pp. 218–229. ACM Press (May 1987)
18. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM 43(3), 431–473 (1996)
19. Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: How to run turing machines on encrypted data. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 536–553. Springer, Heidelberg (Aug 2013)
20. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: CCS (2012)
21. Goyal, V., Lee, C.K., Ostrovsky, R., Visconti, I.: Constructing non-malleable commitments: A black-box approach. In: 53rd FOCS. pp. 51–60. IEEE Computer Society Press (Oct 2012)

22. Goyal, V., Mohassel, P., Smith, A.: Efficient two party and multi party computation against covert adversaries. In: EUROCRYPT. pp. 289–306 (2008)
23. Goyal, V., Ostrovsky, R., Scafuro, A., Visconti, I.: Black-box non-black-box zero knowledge. In: Shmoys, D.B. (ed.) 46th ACM STOC. pp. 515–524. ACM Press (May / Jun 2014)
24. Groth, J., Ostrovsky, R., Sahai, A.: Perfect non-interactive zero knowledge for np. In: Proceedings of Eurocrypt 2006, volume 4004 of LNCS. pp. 339–358. Springer (2006)
25. Hazay, C., Yanai, A.: Constant-round maliciously secure two-party computation in the RAM model. In: TCC (2016-B)
26. Impagliazzo, R., Rudich, S.: Limits on the provable consequences of one-way permutations. In: 21st ACM STOC. pp. 44–61. ACM Press (May 1989)
27. Impagliazzo, R., Rudich, S.: Limits on the provable consequences of one-way permutations. In: Goldwasser, S. (ed.) CRYPTO'88. LNCS, vol. 403, pp. 8–26. Springer, Heidelberg (Aug 1990)
28. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (Aug 2003)
29. Ishai, Y., Kushilevitz, E., Lindell, Y., Petrank, E.: Black-box constructions for secure computation. In: Kleinberg, J.M. (ed.) 38th ACM STOC. pp. 99–108. ACM Press (May 2006)
30. Ishai, Y., Prabhakaran, M., Sahai, A.: Founding cryptography on oblivious transfer - efficiently. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 572–591. Springer, Heidelberg (Aug 2008)
31. Lin, H., Pass, R.: Succinct garbling schemes and applications. Cryptology ePrint Archive, Report 2014/766 (2014), http://eprint.iacr.org/2014/766
32. Lu, S., Ostrovsky, R.: How to garble RAM programs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 719–734. Springer, Heidelberg (May 2013)
33. Miao, P.: Cut-and-choose for garbled RAM. Personal Communication (2016)
34. Nielsen, J.B., Orlandi, C.: LEGO for two-party secure computation. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 368–386. Springer, Heidelberg (Mar 2009)
35. Ostrovsky, R.: Efficient computation on oblivious RAMs. In: 22nd ACM STOC. pp. 514–523. ACM Press (May 1990)
36. Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: 29th ACM STOC. pp. 294–303. ACM Press (May 1997)
37. Pass, R., Wee, H.: Black-box constructions of two-party protocols from one-way functions. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 403–418. Springer, Heidelberg (Mar 2009)
38. Pippenger, N., Fischer, M.J.: Relations among complexity measures. J. ACM 26(2), 361–381 (1979)
39. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) 37th ACM STOC. pp. 84–93. ACM Press (May 2005)
40. Wee, H.: Black-box, round-efficient secure computation via non-malleability amplification. In: 51st FOCS. pp. 531–540. IEEE Computer Society Press (Oct 2010)
41. Yao, A.C.C.: Protocols for secure computations (extended abstract). In: 23rd FOCS. pp. 160–164. IEEE Computer Society Press (Nov 1982)