Adaptive Garbled RAM from Laconic Oblivious Transfer

Sanjam Garg[⋆1], Rafail Ostrovsky[⋆⋆2], and Akshayaram Srinivasan[1]

[1] University of California, Berkeley
{sanjamg,akshayaram}@berkeley.edu
[2] UCLA
rafail@cs.ucla.edu

Abstract. We give a construction of an adaptive garbled RAM scheme. In the adaptive setting, a client first garbles a "large" persistent database which is stored on a server. Next, the client can provide garbling of multiple adaptively and adversarially chosen RAM programs that execute and modify the stored database arbitrarily. The garbled database and the garbled program should reveal nothing more than the running time and the output of the computation. Furthermore, the sizes of the garbled database and the garbled program grow only linearly in the size of the database and the running time of the executed program respectively (up to poly logarithmic factors). The security of our construction is based on the assumption that laconic oblivious transfer (Cho et al., CRYPTO 2017) exists. Previously, such adaptive garbled RAM constructions were only known using indistinguishability obfuscation or in random oracle model. As an additional application, we note that this work yields the first constant round secure computation protocol for persistent RAM programs in the malicious setting from standard assumptions. Prior works did not support persistence in the malicious setting.

1 Introduction

Over the years, garbling methods [Yao86,LP09,AIK04,BHR12b,App17] have been extremely influential and have engendered an enormous number of applications

⋆ Research supported in part from DARPA/ARL SAFEWARE Award W911NF15C0210, AFOSR Award FA9550-15-1-0274, AFOSR YIP Award, DARPA and SPAWAR under contract N66001-15-C-4065, a Hellman Award and research grants by the Okawa Foundation, Visa Inc., and Center for Long-Term Cybersecurity (CLTC, UC Berkeley). The views expressed are those of the author and do not reflect the official policy or position of the funding agencies.

⋆⋆ Research supported in part by NSF grant 1619348, DARPA SPAWAR contract N66001-15-1C-4065, US-Israel BSF grant 2012366, OKAWA Foundation Research Award, IBM Faculty Research Award, Xerox Faculty Research Award, B. John Garrick Foundation Award, Teradata Research Award, and Lockheed-Martin Corporation Research Award. The views expressed are those of the authors and do not reflect position of the Department of Defense or the U.S. Government.

in cryptography. Informally, garbling a function f and an input x, yields the function encoding $\widehat{f}$ and the input encoding $\widehat{x}$. Given $\widehat{f}$ and $\widehat{x}$, there exists an efficient decoding algorithm that recovers $f(x)$. The security property requires that $\widehat{f}$ and $\widehat{x}$ do not reveal anything about f or x except $f(x)$. By now, it is well established that realizing garbling schemes [BHR12b,App17] is an important cryptographic goal.

One shortcoming of standard garbling techniques has been that the size of the function encoding grows linearly in the size of the circuit computing the function and thus leads to large communication costs. Several methods have been devised to overcome this constraint.

– Lu and Ostrovsky [LO13] addressed the question of garbling RAM program execution on a persistent garbled database. Here, the efficiency requirement is that the size of the function encoding grows only with the running time of the RAM program. This work has lead to fruitful line of research [GHL$^+$14,GLOS15,GLO15,LO17] that reduces the communication cost to grow linearly with running times of the programs executed, rather that the corresponding circuit sizes. A key benefit of this approach is that it has led to constructions based on one-way functions.
– Goldwasser, Kalai, Popa, Vaikuntanathan, and Zeldovich [GKP$^+$13] addressed the question of reducing the communication cost by reusing the encodings. Specifically, they provided a construction of reusable garbled circuits based on standard assumptions (namely learning-with-errors). However, their construction needs input encoding to grow with the depth of the circuit being garbled.
– Finally, starting with Gentry, Halevi, Raykova, and Wichs [GHRW14], a collection of works [CHJV15,BGL$^+$15,KLW15,CH16,CCHR16,ACC$^+$16] have attempted to obtain garbling schemes where the size of the function encoding only grows with its description size and is otherwise independent of its running time on various inputs. However, these constructions are proven secure only assuming indistinguishability obfuscation [BGI$^+$01,GGH$^+$13].

A recurring theme in all the above research efforts has been the issue of *adaptivity*: Can the adversary adaptively choose the input after seeing the function encoding?

This task is trivial if one reveals both the function encoding and the input encoding together after the input is specified. However, this task becomes highly non-trivial if we require the size of the input encoding to only grow with the size of the input and independent of the complexity of computing f. The first solution to this problem was provided by Bellare, Hoang and Rogaway [BHR12a] for the case of circuits in the random oracle model [BR93]. Subsequently, several adaptive circuit garbling schemes have been obtained in the standard model from (i) one-way functions [HJO$^+$16,JW16,JKK$^+$17],[3] or (ii) using laconic OT [GS18a] which relies on public-key assumptions [CDG$^+$17,DG17,DGHM18,BLSV18].

[3] A drawback of these works is that the size of the input encoding grows with the width/depth of the circuit computing f

However, constructing adaptively secure schemes for more communication constrained settings has proved much harder. In this paper, we focus on the case of RAM programs. More specifically, adaptively secure garbled RAM is known only using random oracles (e.g. [LO13,GLOS15]) or under very strong assumptions such as indistinguishability obfuscation [CCHR16,ACC$^+$16]. In this work, we ask:

Can we realize adaptively secure garbled RAM from standard assumptions?

Further motivating the above question, is the tightly related application of constructing *constant round* secure RAM computation over a persistent database in the malicious setting. More specifically, as shown by Beaver, Micali and Rogaway [BMR90] garbling techniques can be used to realize constant round secure computation [Yao82,GMW87] constructions. Similarly, above-mentioned garbling schemes for RAM programs also yield constant round, communication efficient secure computation solutions [HY16,Mia16,GGMP16,KY18]. However, preserving persistence of RAM programs in the malicious setting requires the underlying garbling techniques to provide adaptive security.[4]

1.1 Our Results

In this work, we obtain a construction of adaptively secure garbled RAM based on the assumption that laconic oblivious transfer [CDG$^+$17] exists. Laconic oblivious transfer can be based on a variety of public-key assumptions such as (i) Computation Diffie-Hellman Assumption [DG17], (ii) Factoring Assumption [DG17], or (iii) Learning-With-Errors Assumption [BLSV18,DGHM18]. In our construction, the size of the garbled database and the garbled program grow only linearly in the size of the database and the running time of the executed program respectively (up to poly logarithmic factors). The main result in our paper is:

Theorem 1 (Informal). *Assuming either the Computational Diffie-Hellman assumption or the Factoring assumption or the Learning-with-Errors assumption, there exists a construction of adaptive garbled RAM scheme where the time required to garble a database, a program and an input grows linearly (upto poly logarithmic factors) with the size of the database, running time of the program and length of the input respectively.*[5]

Additionally, plugging our adaptively secure garbled RAM scheme into a malicious secure constant round secure computation protocol yields a maliciously secure constant round secure RAM computation protocol [IKO$^+$11,ORS15,BL18,GS18b]

[4] We note that adaptive security is not essential for obtaining protocols with round complexity that grows with the running time of the executed programs [OS97,GKK$^+$12,WHC$^+$14].

[5] As in the case of adaptively secure garbled circuits, the size of the input encoding must also grow with the output length of the program. Here, we implicitly assume that the input and the outputs have the same length.

for a persistent database. Again, this construction is based on the assumption that laconic OT exists and the underlying assumptions needed for the constant round protocol.

2 Our Techniques

In this section, we outline the main challenges and the techniques used in our construction of adaptive garbled RAM.

Starting Point. In a recent result, Garg and Srinivasan [GS18a] gave a construction of adaptively secure garbled circuit transfer where the size of the input encoding grows only with the input and the output length. The main idea behind their construction is a technique to "linearize" a garbled circuit. Informally, a garbled circuit is said to be linearized if the simulation of particular garbled gate depends only on simulating one other gate (or in other words, the simulation dependency graph is a line). In order to linearize a garbled circuit, their work transforms a circuit into a sequence of CPU step circuits that can make read and write accesses at *fixed* locations in an external memory. The individual step circuits are garbled using a (plain) garbling scheme and the access to the memory is mediated using a laconic OT.[6] The use of laconic OT enables the above mentioned garbling scheme to have "linear" structure wherein the simulation of a particular CPU step depends only on simulating the previous step circuit.

A Generalization. Though the approach of Garg and Srinivasan shares some similarities with a garbling a RAM program (like garbling a sequence of CPU step circuits), there are some crucial differences.

1. The first difference is that unlike a circuit, the locations that are accessed by a RAM program are dynamically chosen depending on the program's input.
2. The second difference is that the locations that are accessed might leak information about the program and the input and a garbled RAM scheme must protect against such leakages.

The first step we take in constructing an adaptive garbled RAM scheme is to generalize the above approach of Garg and Srinivasan [GS18a] to construct an adaptively secure garbled RAM scheme with weaker security guarantees. The security that we achieve is that of unprotected memory access [GHL$^+$14]. Informally, a garbled RAM scheme is said to have unprotected memory access if both the contents of the database and the memory locations that are accessed are revealed in the clear. This generalization is given in Section 4.

[6] A laconic OT scheme allows to compress a large database/memory to a small digest. The digest in some sense binds the entire database. In particular, given the digest there exists efficient algorithms that can read/update particular memory locations. The time taken by these algorithms grow only logarithmically with the size of the database.

In the non-adaptive setting, there are standard transformations (outlined in [GHL$^+$14]) from a garbled RAM with unprotected memory access to a standard garbled RAM scheme where both the memory contents and the access patterns are hidden. This transformation involves the additional use of an ORAM scheme. Somewhat surprisingly, these transformations fail in the adaptive setting! The details follow.

Challenges. To understand the main challenges, let us briefly explain how the security proof goes through in the work of Garg and Srinivasan [GS18a]. In a typical construction of a garbled RAM program, using a sequence of garbled circuits, one would expect that the simulation of garbled circuits would be done from the first CPU step to the last CPU step. However, in [GS18a] proof, the simulation is done in a rather unusual manner, from the last CPU step to the first CPU step. Of course, it is not possible to simulate the last CPU step directly. Thus, the process of simulating the last CPU step itself involves a sequence of hybrids that simulate and "un-simulate" the garbling of the previous CPU steps. Extending this approach so that the memory contents and the access patterns are both hidden faces the following two main challenges.

- **Challenge 1:** In the Garg and Srinivasan construction [GS18a], memory contents were encrypted using one-time pads. Since the locations that each CPU step (for a circuit) reads from and write to are fixed, the one-time pad corresponding to that location could be hardwired to those CPU steps. On the other hand, in the case of RAM programs the locations being accessed are dynamically chosen and thus it is not possible to hard-wire the entire one-time pad into each CPU step as this would blow up the size of these CPU steps.
  It is instructive to note that encrypting the memory using an encryption scheme and decrypting the read memory contents does not suffice. See more on this in preliminary attempt below.
- **Challenge 2:** In the non-adaptive setting, it is easy to amplify unprotected memory access security to the setting where memory accesses are hidden using an oblivious RAM scheme [Gol87,Ost90,GO96]. However, in the adaptive setting this transformation turns out to be tricky. In a bit more detail, the Garg and Srinivasan [GS18a] approach of simulating CPU step circuits from the last to the first ends up in conflict with the security of the ORAM scheme where the simulation is typically done from the first to the last CPU steps. We note here that the techniques of Canetti et al. [CCHR16] and Ananth et al. [ACC$^+$16], though useful, do not apply directly to our setting. In particular, in the Canetti et al. [CCHR16] and Ananth et al. [ACC$^+$16] constructions, CPU steps where obfuscated using an indistinguishability obfuscation scheme. Thus, in their scheme the obfuscation for any individual CPU step could be changed independently. For example, the PRF key used in any CPU step could be punctured independent of the other CPU steps. On the other hand, in our construction, inspite of each CPU step being garbled separately, its input labels are hardwired in the previous garbled circuit.

Therefore, a change in hardwired secret value (like a puncturing a key) in a CPU step needs an intricate sequence of hybrids for making this change. For instance, in the case of the example above, it is not possible to puncture the PRF key hardwired in a particular CPU step in one simple hybrid step. Instead any change in this CPU step must change the CPU step before it and so on. In summary, in our case, any such change would involve a new and intricate hybrid argument.

2.1 Solving Challenge 1

In this subsection, we describe our techniques to solve challenge 1.

Preliminary Attempt. A very natural approach to encrypting external memory would be to use a pseudorandom function to encrypt memory content in each location. More precisely, a data value d in location L is encrypted using the key $\mathsf{PRF}_K(L)$ where K is the PRF key. The key K for this pseudorandom function is hardwired in each CPU step so that it first decrypts the ciphertext that is read from the memory and uses the underlying data for further processing. This approach to solving Challenge 1 was in fact used in the works of Canetti et al. [CCHR16] and Ananth et al. [ACC$^+$16] (and several other prior works) in a similar context. However, in order to use the security of this PRF, we must first remove the hardwired key from each of the CPU steps. This is easily achieved if we rely on indistinguishability obfuscation. Indeed, a single hybrid change is sufficient to have the punctured key to be hardwired in each of the CPU steps. However, in our setting this does not work! In particular, we need to puncture the PRF key in each of the CPU step circuits by simulating them individually and the delicate dependencies involved in garbling each CPU step blows up the size of the garbled input to grow with the running time of the program.[7] Due to the same reason, the approaches of encrypting the memory by maintaining a tree of secret keys [GLOS15,GLO15] do not work.

Our New Idea: A Careful Timed Encryption Mechanism. From the above attempts, the following aspect of secure garbled RAM arise. Prior approaches for garbling RAM programs use PRF keys that in some sense "decrease in power"[8] as hybrids steps involve sequential simulation of the CPU steps starting with the first CPU step and ending in the last CPU step. However, in the approach of [GS18a], the hybrids do a backward pass, from the last CPU step circuit to the first CPU step circuit. Therefore, we need a mechanism wherein

[7] For the readers who are familiar with [GS18a], the number of CPU steps that have to be maintained in the input dependent simulation for puncturing the PRF key grows with the number of CPU steps that last wrote to this location and this could be as large as the running time of the program.

[8] The tree-based approaches of storing the secret keys use the mechanism wherein the hardwired secret keys decrease in power in subsequent CPU steps. In particular, the secret key corresponding to the root can decrypt all the locations, the secret keys corresponding to its children can only decrypt a part of the database and so on.

the hardwired key for encryption in some sense "strengthens" along the first to the last CPU step.

Location vs. Time. In almost all garbled RAM constructions, the data stored at a particular location is encrypted using a location dependent key (e.g. [GLOS15]). This was not a problem when the keys are being weakened across CPU steps. However, in our case we need the key to be strengthened in power across CPU steps. Thus, we need a special purpose encryption scheme where the keys are derived based on time rather than the locations. Towards this goal, we construct a special purpose encryption scheme called as a *timed encryption* scheme. Let us explain this in more detail.

Timed Encryption. A timed encryption scheme is just like any (plain) symmetric key encryption except that every message is encrypted with respect to a timestamp. Additionally, there is a special key constrain algorithm that constrains a key to only decrypt ciphertexts that are encrypted within a specific timestamp. The security requirement is that the constrained key does not help in distinguishing ciphertexts of two messages that are encrypted with respect to some future timestamp. We additionally require the encryption using a key constrained with respect to a timestamp time to have the same distribution as an encryption using an unconstrained key as long as the timestamp to which we are encrypting is less than or equal to time. For efficiency, we require that the size of the constrained key to grow only with the length of the binary representation of the timestamp.

Solving Challenge 1. Timed encryption provides a natural approach to solving challenge 1. In every CPU step, we hardwire a time constrained key that allows that CPU step to decrypt all the memory updates done by the prior CPU steps. The last CPU step in some sense has the most powerful key hardwired, i.e., it can decrypt all the updates made by all the prior CPU steps and the first CPU step has the least powerful key hardwired. Thus, the hardwired secret key strengthens from the first CPU step to the last CPU step. In the security proof, a backward pass of simulating the last CPU step to the first CPU step conforms well with the semantics and security properties of a timed encryption scheme. This is because we remove the most powerful keys first and the rest of the hardwired secret keys in the previous CPU steps do not help in distinguishing between encryptions of the actual value that is written and some junk value. We believe that the notion timed encryption might have other applications and be of independent interest.

Constructing Timed Encryption. We give a construction of a timed encryption scheme from any one-way function. Towards this goal, we introduce a notion called as *range constrained PRF*. A range constrained PRF is a special constrained PRF [BW13] where the PRF key can be constrained to evaluate input points that fall within a particular range. The ranges that we will be interested in are of the form $[0, x]$. That is, the constrained key can be used to evaluate the PRF on any $y \in [0, x]$. For efficiency, we require that the size of the constrained key to only grow with the binary representation of x. Given such a PRF, we can construct a timed encryption scheme as follows. The key generation samples a range constrained PRF key. The encryption of a message m with respect to a timestamp time proceeds by evaluating the PRF on time to derive sk and then using sk as a key for symmetric encryption scheme to encrypt the message m. The time constraining algorithm just constrains the PRF key with respect to the range $[0, \mathsf{time}]$. Thus, the goal of constructing a timed encryption scheme reduces to the goal of constructing a range constrained PRF. In this work, we give a construction of range constrained PRF by adding a range constrain algorithm to the tree-based PRF scheme of Goldreich, Goldwasser and Micali [GGM86].

2.2 Solving Challenge 2

Challenge 1 involves protecting the contents of the memory whereas challenge 2 involves protecting the access pattern. As mentioned before, in the non-adaptive setting, this problem is easily solved using an oblivious RAM scheme. However, in our setting we need an oblivious RAM scheme with some special properties.

The works of Canetti et al. [CCHR16] and Ananth et al. [ACC^{+}16] define a property of an ORAM scheme as *strong localized randomness property* and then use this property to hide their access patterns. Informally, an ORAM scheme is said to have a strong localized randomness property if the locations of the random tape accessed by an oblivious program in simulating each memory access are disjoint. Further, the number of locations touched for simulating each memory access must be poly logarithmic in the size of the database. These works further proved that the Chung-Pass ORAM scheme [CP13] satisfies the strong localized randomness property. Unfortunately, this strong localized randomness property alone is not sufficient for our purposes. Let us give the details.

To understand why the strong localized randomness property alone is not sufficient, we first recall the details of the Chung-Pass ORAM (henceforth, denoted as CP ORAM) scheme. The CP ORAM is a tree-based ORAM scheme where the leaves of this tree are associated with the actual memory. A position map associates each data block in the memory with a random leaf node. Accessing a memory location involves first reading the position map to get the address of the leaf where this data block resides. Then, the path from the root to this particular leaf is traversed and the content of the this data block is read. It is guaranteed that the data block is located somewhere along the path from the root to leaf node. The read data block is then placed in the root and the position map is updated so that another random leaf node is associated with this data block. To balance the memory, an additional flush is performed but for the sake of this introduction we ignore this step. The CP ORAM scheme has strong localized randomness as the randomness used in each memory accesses involves choosing a random leaf to update the position map. Let us now explain why this property alone is not sufficient for our purpose.

Recall that in the security proof of [GS18a], the CPU steps are simulated from the last step to the first. A simulation of a CPU step involves changing the bit written by the step to some junk value and the changing the location accessed to a random location. We can change the bit to be written to a junk value using the security of the timed encryption scheme, however changing the location accessed to random is problematic. Note that the location that is being accessed in the CP ORAM is a random root to leaf path. However, the address of this leaf is stored in the memory via the position map. Therefore, to simulate a particular CPU step, we must first change the contents of the position map. This change must be performed in those CPU steps that last updated this memory location. Unfortunately, timed encryption is not useful in this setting as we can use its security only after removing all the secret keys that are hardwired in the future time steps. However, in our case, the CPU steps that last updated this particular location might be so far into the past that removing all the intermediate encryption keys might blow up the cost of the input encoding to be as large as the program running time.

To solve this issue, we modify the Chung-Pass ORAM to additionally have the CPU steps to encrypt the data block that is written using a puncturable PRF. Unlike the previous approaches of encrypting the data block with respect to the location, we encrypt it with respect to the time step that modifies the location. This helps in circumventing the above problem as we can first puncture the PRF key (which in turn involves a careful set of hybrids) and use its security to change the position map to contain an encryption of the junk value instead of the actual address of the leaf node.[9] Once this change is done, the locations that the concerned CPU step is accessing is a random root to leaf path.

3 Preliminaries

Let λ denote the security parameter. A function $\mu(\cdot) : \mathbb{N} \to \mathbb{R}^+$ is said to be negligible if for any polynomial $\mathsf{poly}(\cdot)$ there exists $\lambda_0 \in \mathbb{N}$ such that for all $\lambda > \lambda_0$ we have $\mu(\lambda) < \frac{1}{\mathsf{poly}(\lambda)}$. For a probabilistic algorithm A, we denote $A(x; r)$ to be the output of A on input x with the content of the random tape being r. When r is omitted, $A(x)$ denotes a distribution. For a finite set S, we denote $x \leftarrow S$ as the process of sampling x uniformly from the set S. We will use PPT to denote Probabilistic Polynomial Time. We denote $[a]$ to be the set $\{1, \ldots, a\}$ and $[a, b]$ to be the set $\{a, a + 1, \ldots, b\}$ for $a \leq b$ and $a, b \in \mathbb{Z}$. For a binary string $x \in \{0, 1\}^n$, we will denote the i^{th} bit of x by x_i. We assume without loss of generality that the length of the random tape used by all cryptographic algorithms is λ. We will use $\mathsf{negl}(\cdot)$ to denote an unspecified negligible function and $\mathsf{poly}(\cdot)$ to denote an unspecified polynomial function.

We assume reader's familiarity with the notions of a puncturable PRF and selectively secure garbled circuits and omit the formal definitions here for the lack of space.

3.1 Updatable Laconic Oblivious Transfer

In this subsection, we recall the definition of updatable laconic oblivious transfer from [CDG+17].

[9] Unlike in the location based encryption scheme, it is sufficient to change the encryption only in the CPU steps that last modified this location.

We give the formal definition below from [CDG+17]. We generalize their definition to work for blocks of data instead of bits. More precisely, the reads and the updates happen at the block-level rather than at the bit-level.

Definition 1 ([CDG+17]). *An updatable laconic oblivious transfer consists of the following algorithms:*

- $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda, 1^N)$: *It takes as input the security parameter* 1^λ *(encoded in unary) and a block size* N *and outputs a common reference string* crs.
- $(\mathsf{d}, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$: *It takes as input the common reference string* crs *and database* $D \in \{\{0,1\}^N\}^*$ *as input and outputs a digest* d *and a state* $\widehat{D}$. *We assume that the state* $\widehat{D}$ *also includes the database* D.
- $e \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{d}, L, \{m_{i,0}, m_{i,1}\}_{i\in[N]})$: *It takes as input the common reference string* crs*, a digest* d*, and a location* $L \in \mathbb{N}$ *and set of messages* $m_{i,0}, m_{i,1} \in \{0,1\}^{p(\lambda)}$ *for every* $i \in [N]$ *and outputs a ciphertext* e.
- $(m_1, \ldots, m_N) \leftarrow \mathsf{Receive}^{\widehat{D}}(\mathsf{crs}, e, L)$: *This is a RAM algorithm with random read access to* $\widehat{D}$. *It takes as input a common reference string* crs*, a ciphertext* e*, and a location* $L \in \mathbb{N}$ *and outputs a set of messages* $m_1, \ldots, m_N$.
- $e_w \leftarrow \mathsf{SendWrite}(\mathsf{crs}, \mathsf{d}, L, \{b_i\}_{i\in[N]}, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{d}|})$: *It takes as input the common reference string* crs*, a digest* d*, and a location* $L \in \mathbb{N}$*, bits* $b_i \in \{0,1\}$ *for each* $i \in [N]$ *to be written, and* $|\mathsf{d}|$ *pairs of messages* $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{d}|}$*, where each* $m_{j,c}$ *is of length* $p(\lambda)$ *and outputs a ciphertext* e_w.
- $\{m_j\}_{j=1}^{|\mathsf{d}|} \leftarrow \mathsf{ReceiveWrite}^{\widehat{D}}(\mathsf{crs}, L, \{b_i\}_{i\in[N]}, e_w)$: *This is a RAM algorithm with random read/write access to* $\widehat{D}$. *It takes as input the common reference string* crs*, a location* L*, a set of bits* $b_1, \ldots, b_N \in \{0,1\}$ *and a ciphertext* e_w. *It updates the state* $\widehat{D}$ *(such that* $D[L] = b_1 \ldots b_N$*) and outputs messages* $\{m_j\}_{j=1}^{|\mathsf{d}|}$.

We require an updatable laconic oblivious transfer to satisfy the following properties.

Correctness: *We require that for any database* D *of size at most* $M = \mathsf{poly}(\lambda)$*, any memory location* $L \in [M]$*, any set of messages* $(m_{i,0}, m_{i,1}) \in \{0,1\}^{p(\lambda)}$ *for each* $i \in [N]$ *where* $p(\cdot)$ *is a polynomial that*

$$\Pr\left[\forall i \in [N],\ m_i = m_{i,D[L,i]} \;\middle|\; \begin{array}{cl} \mathsf{crs} & \leftarrow \mathsf{crsGen}(1^\lambda) \\ (\mathsf{d}, \widehat{D}) & \leftarrow \mathsf{Hash}(\mathsf{crs}, D) \\ e & \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{d}, L, \{m_{i,0}, m_{i,1}\}_{i\in[N]}) \\ (m_1, \ldots, m_N) & \leftarrow \mathsf{Receive}^{\widehat{D}}(\mathsf{crs}, e, L) \end{array}\right] = 1,$$

where $D[L,i]$ *denotes the* i^{th} *bit in the* L^{th} *block of* D.

Correctness of Writes: *Let database* D *be of size at most* $M = \mathsf{poly}(\lambda)$ *and let* $L \in [M]$ *be any two memory locations. Let* D^* *be a database that is identical to* D *except that* $D^*[L,i] = b_i$ *for all* $i \in [N]$ *some sequence of* $\{b_j\} \in \{0,1\}$.

For any sequence of messages $\{m_{j,0}, m_{j,1}\}_{j\in[\lambda]} \in \{0,1\}^{p(\lambda)}$ *we require that*

$$\Pr\left[\begin{array}{c|rcl} & \mathsf{crs} & \leftarrow & \mathsf{crsGen}(1^\lambda, 1^N) \\ m'_j = m_{j,\mathsf{d}^*_j} & (\mathsf{d}, \widehat{D}) & \leftarrow & \mathsf{Hash}(\mathsf{crs}, D) \\ \forall j \in [|\mathsf{d}|] & (\mathsf{d}^*, \widehat{D}^*) & \leftarrow & \mathsf{Hash}(\mathsf{crs}, D^*) \\ & e_w & \leftarrow & \mathsf{SendWrite}(\mathsf{crs}, \mathsf{d}, L, \{b_i\}_{i\in[N]}, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{d}|}) \\ & \{m'_j\}_{j=1}^{|\mathsf{d}|} & \leftarrow & \mathsf{ReceiveWrite}^{\widehat{D}}(\mathsf{crs}, L, \{b_i\}_{i\in[N]}, e_w) \end{array}\right] = 1,$$

Sender Privacy: *There exists a PPT simulator* $\mathsf{Sim}_{\ell\mathsf{OT}}$ *such that the for any non-uniform PPT adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *there exists a negligible function* $\mathsf{negl}(\cdot)$ *s.t.,*

$$\left| \Pr[\mathsf{Expt}^{\mathsf{real}}(1^\lambda, \mathcal{A}) = 1] - \Pr[\mathsf{Expt}^{\mathsf{ideal}}(1^\lambda, \mathcal{A}) = 1] \right| \leq \mathsf{negl}(\lambda)$$

where $\mathsf{Expt}^{\mathsf{real}}$ *and* $\mathsf{Expt}^{\mathsf{ideal}}$ *are described in Figure 1.*

$\mathsf{Expt}^{\mathsf{real}}[1^\lambda, \mathcal{A}]$

1. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda, 1^N)$.
2. $(D, L, \{m_{i,0}, m_{i,1}\}_{i\in[N]}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{crs})$.
3. $(\mathsf{d}, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$.
4. Output $\mathcal{A}_2(\mathsf{st}, \mathsf{Send}(\mathsf{crs}, \mathsf{d}, L, \{m_{i,0}, m_{i,1}\}_{i\in[N]}))$.

$\mathsf{Expt}^{\mathsf{ideal}}[1^\lambda, \mathcal{A}]$

1. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$.
2. $(D, L, \{m_{i,0}, m_{i,1}\}_{i\in[N]}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{crs})$.
3. $(\mathsf{d}, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$.
4. Output $\mathcal{A}_2(\mathsf{st}, \mathsf{Sim}_{\ell\mathsf{OT}}(\mathsf{crs}, D, L, \{m_{i,D[L,i]}\}_{i\in[N]}))$.

Figure 1: Sender Privacy Security Game

Sender Privacy for Writes: *There exists a PPT simulator* $\mathsf{Sim}_{\ell\mathsf{OTW}}$ *such that the for any non-uniform PPT adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *there exists a negligible function* $\mathsf{negl}(\cdot)$ *s.t.,*

$$\left| \Pr[\mathsf{WriSenPrivExpt}^{\mathsf{real}}(1^\lambda, \mathcal{A}) = 1] - \Pr[\mathsf{WriSenPrivExpt}^{\mathsf{ideal}}(1^\lambda, \mathcal{A}) = 1] \right| \leq \mathsf{negl}(\lambda)$$

where $\mathsf{WriSenPrivExpt}^{\mathsf{real}}$ *and* $\mathsf{WriSenPrivExpt}^{\mathsf{ideal}}$ *are described in Figure 2.*

Efficiency: *The algorithm Hash runs in time* $|D|\mathsf{poly}(\log|D|, \lambda)$. *The algorithms* Send, $\mathsf{SendWrite}$, $\mathsf{Receive}$, $\mathsf{ReceiveWrite}$ *run in time* $N \cdot \mathsf{poly}(\log|D|, \lambda)$.

Theorem 2 ([CDG$^+$17,DG17,BLSV18,DGHM18]). *Assuming either the Computational Diffie-Hellman assumption or the Factoring assumption or the Learning with Errors assumption, there exists a construction of updatable laconic oblivious transfer.*

Remark 1. We note that the security requirements given in Definition 1 is stronger than the one in [CDG$^+$17] as we require the crs to be generated before the adversary provides the database D and the location L. However, the construction in [CDG$^+$17] already satisfies this definition since in the proof, we can guess the location by incurring a $1/|D|$ loss in the security reduction.

```
$\mathsf{WriSenPrivExpt}^{\mathsf{real}}[1^\lambda, \mathcal{A}]$

1. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda, 1^N)$.
2. $(D, L, \{b_i\}_{i\in[N]}, \{m_{j,0}, m_{j,1}\}_{j\in[\lambda]}, \mathsf{st})$
   $\leftarrow \mathcal{A}_1(\mathsf{crs})$.
3. $(\mathsf{d}, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$.
4. $e_w \leftarrow \mathsf{SendWrite}(\mathsf{crs}, \mathsf{d}, L, \{b_i\}_{i\in[N]},$
   $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{d}|})$
5. Output $\mathcal{A}_2(\mathsf{st}, e_w)$.
```

```
$\mathsf{WriSenPrivExpt}^{\mathsf{ideal}}[1^\lambda, \mathcal{A}]$

1. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda, 1^N)$.
2. $(D, L, \{b_i\}_{i\in[N]}, \{m_{j,0}, m_{j,1}\}_{j\in[\lambda]}, \mathsf{st})$
   $\leftarrow \mathcal{A}_1(\mathsf{crs})$.
3. $(\mathsf{d}, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$.
4. $(\mathsf{d}^*, \widehat{D}^*) \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*)$ where $D^*$
   be a database that is identical to $D$
   except that $D^*[L, i] = b_i$ for each $i \in$
   $[N]$.
5. $e_w \leftarrow \mathsf{Sim}_{\ell\mathsf{OTW}}(\mathsf{crs}, D, L, \{b_i\}_{i\in[N]},$
   $\{m_{j,\mathsf{d}^*_j}\}_{j\in[\lambda]})$
6. Output $\mathcal{A}_2(\mathsf{st}, e_w)$.
```

Figure 2: Sender Privacy for Writes Security Game

3.2 Somewhere Equivocal Encryption

We now recall the definition of Somewhere Equivocal Encryption from the work of [HJO+16]. Informally, a somewhere equivocal encryption allows to create a simulated ciphertext encrypting a message m with certain positions of the message being "fixed" and the other positions having a "hole." The simulator can later fill these "holes" with arbitrary message values by deriving a suitable decryption key. The main efficiency requirement is that the size of the decryption key grows only with the number of "holes" and is otherwise independent of the message size. We give the formal definition below.

Definition 2 ([HJO+16]). *A somewhere equivocal encryption scheme with block-length s, message length n (in blocks) and equivocation parameter t (all polynomials in the security parameter) is a tuple of probabilistic polynomial algorithms $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{SimEnc}, \mathsf{SimKey})$ such that:*

- $\mathsf{key} \leftarrow \mathsf{KeyGen}(1^\lambda)$: *It is a PPT algorithm that takes as input the security parameter (encoded in unary) and outputs a key* key.
- $\overline{c} \leftarrow \mathsf{Enc}(\mathsf{key}, m_1 \ldots m_n)$: *It is a PPT algorithm that takes as input a key* key *and a vector of messages $\overline{m} = m_1 \ldots m_n$ with each $m_i \in \{0,1\}^s$ and outputs a ciphertext $\overline{c}$.*
- $\overline{m} \leftarrow \mathsf{Dec}(\mathsf{key}, \overline{c})$: *It is a deterministic algorithm that takes as input a key* key *and a ciphertext $\overline{c}$ and outputs a vector of messages $\overline{m} = m_1 \ldots m_n$.*
- $(\mathsf{st}, \overline{c}) \leftarrow \mathsf{SimEnc}((m_i)_{i\notin I}, I)$: *It is a PPT algorithm that takes as input a set of indices $I \subseteq [n]$ and a vector of messages $(m_i)_{i\notin I}$ and outputs a ciphertext $\overline{c}$ and a state* st.
- $\mathsf{key}' \leftarrow \mathsf{SimKey}(\mathsf{st}, (m_i)_{i\in I})$: *It is a PPT algorithm that takes as input the state information* st *and a vector of messages $(m_i)_{i\in I}$ and outputs a key* key'.

and satisfies the following properties:

Correctness. *For every* $\mathsf{key} \leftarrow \mathsf{KeyGen}(1^\lambda)$*, for every* $\overline{m} \in \{0,1\}^{s \times n}$ *it holds that:*

$$\mathsf{Dec}(\mathsf{key}, \mathsf{Enc}(\mathsf{key}, \overline{m})) = \overline{m}$$

Simulation with No Holes. *We require that the distribution of* $(\overline{c}, \mathsf{key})$ *computed via* $(\mathsf{st}, \overline{c}) \leftarrow \mathsf{SimEnc}(\overline{m}, \emptyset)$ *and* $\mathsf{key} \leftarrow \mathsf{SimKey}(\mathsf{st}, \emptyset)$ *to be identical to* $\mathsf{key} \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and* $\overline{c} \leftarrow \mathsf{Enc}(\mathsf{key}, m_1 \dots m_n)$*. In other words, simulation when there are no holes (i.e.,* $I = \emptyset$*) is identical to honest key generation and encryption.*

Security. *For any PPT adversary* $\mathcal{A}$*, there exists a negligible function* $\nu = \nu(\lambda)$ *such that:*

$$\left| \Pr[\mathsf{Exp}^{\mathsf{simenc}}_{\mathcal{A},\Pi}(1^\lambda, 0) = 1] - \Pr[\mathsf{Exp}^{\mathsf{simenc}}_{\mathcal{A},\Pi}(1^\lambda, 1) = 1] \right| \leq \nu(\lambda)$$

where the experiment $\mathsf{Exp}^{\mathsf{simenc}}_{\mathcal{A},\Pi}$ *is defined as follows:*

Experiment $\mathsf{Exp}^{\mathsf{simenc}}_{\mathcal{A},\Pi}$

1. *The adversary* $\mathcal{A}$ *on input* 1^λ *outputs a set* $I \subseteq [n]$ *s.t.* $|I| < t$*, a vector* $(m_i)_{i \notin I}$*, and a challenge* $j \in [n] \setminus I$*. Let* $I' = I \cup \{j\}$*.*
2. – *If* $b = 0$*, compute* $\overline{c}$ *as follows:* $(\mathsf{st}, \overline{c}) \leftarrow \mathsf{SimEnc}((m_i)_{i \notin I}, I)$*.*
   – *If* $b = 1$*, compute* $\overline{c}$ *as follows:* $(\mathsf{st}, \overline{c}) \leftarrow \mathsf{SimEnc}((m_i)_{i \notin I'}, I')$*.*
3. *Send* $\overline{c}$ *to the adversary* $\mathcal{A}$*.*
4. *The adversary* $\mathcal{A}$ *outputs the set of remaining messages* $(m_i)_{i \in I}$*.*
   – *If* $b = 0$*, compute* key *as follows:* $\mathsf{key} \leftarrow \mathsf{SimKey}(\mathsf{st}, (m_i)_{i \in I})$*.*
   – *If* $b = 1$*, compute* key *as follows:* $\mathsf{key} \leftarrow \mathsf{SimKey}(\mathsf{st}, (m_i)_{i \in I'})$
5. *Send* key *to the adversary.*
6. $\mathcal{A}$ *outputs* b' *which is the output of the experiment.*

Theorem 3 ([HJO+16]). *Assuming the existence of one-way functions, there exists a somewhere equivocal encryption scheme for any polynomial message-length* n*, black-length* s *and equivocation parameter* t*, having key size* $t \cdot s \cdot \mathsf{poly}(\lambda)$ *and ciphertext of size* $n \cdot s \cdot \mathsf{poly}(\lambda)$ *bits.*

3.3 Random Access Machine (RAM) Model of Computation

We start by describing the Random Access Machine (RAM) model of computation in Section 3.3. Most of this subsection is taken verbatim from [CDG+17].

Notation for the RAM Model of Computation. The RAM model consists of a CPU and a memory storage of M blocks where each block has length N. The CPU executes a program that can access the memory by using read/write operations. In particular, for a program P with memory of size M, we denote the initial contents of the memory data by $D \in \{\{0,1\}^N\}^M$. Additionally, the program gets a "short" input $x \in \{0,1\}^n$, which we alternatively think of as the initial state of the program. We use $|P|$ to denote the running time of program P. We use the notation $P^D(x)$ to denote the execution of program P with initial memory contents D and input x. The program P can read from and write to various locations in memory D throughout its execution.[10]

We will also consider the case where several different programs are executed sequentially and the memory persists between executions. We denote this process as $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$ to indicate that first $P_1^D(x_1)$ is executed, resulting in some memory contents D_1 and output y_1, then $P_2^{D_1}(x_2)$ is executed resulting in some memory contents D_2 and output y_2 etc. As an example, imagine that D is a huge database and the programs P_i are database queries that can read and possibly write to the database and are parameterized by some values x_i.

CPU-Step Circuit. Consider an execution of a RAM program which involves at most T CPU steps. We represent a RAM program P via T small *CPU-Step Circuits* each of which executes one CPU step. In this work we will denote one CPU step by:[11]

$$C^P_{\mathsf{CPU}}(\mathsf{state}, \mathsf{rData}) = (\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData})$$

This circuit takes as input the current CPU state state and $\mathsf{rData} \in \{0,1\}^N$. Looking ahead the data rData will be read from the memory location that was requested by the previous CPU step. The circuit outputs an updated state state', a read or write $\mathsf{R/W}$, the next location to read/write from $L \in [M]$, and data wData to write into that location ($\mathsf{wData} = \perp$ when reading). The sequence of locations accessed during the execution of the program collectively form what is known as the *access pattern*, namely $\mathsf{MemAccess} = \{(\mathsf{R/W}^\tau, L^\tau) : \tau = 1, \ldots, T\}$. We assume that the CPU state state contains information about the location that the previous CPU step requested to read from. In particular, $\mathsf{lastLocation}(\mathsf{state})$ outputs the location that the previous CPU step requested to read and it is $\perp$ if the previous CPU step was a write.

Note that in the description above without loss of generality we have made some simplifying assumptions. We assume that each CPU-step circuit always reads from or writes to some location in memory. This is easy to implement via a dummy read and write step. Moreover, we assume that the instructions of the program itself are hardwired into the CPU-step circuits.

Representing RAM computation by CPU-Step Circuits. The computation $P^D(x)$ starts with the initial state set as $\mathsf{state}_1 = x$. In each step $\tau \in$

[10] In general, the distinction between what to include in the program P, the memory data D and the short input x can be somewhat arbitrary. However as motivated by our applications we will typically be interested in a setting where the data D is large while the size of the program $|P|$ and input length x is small.

[11] In the definition below, we model each C_{CPU} as a deterministic circuit. Later, we extend the definition to allow each C_{CPU} to have access to random coins.

$\{1, \ldots T\}$, the computation proceeds as follows: If $\tau = 1$ or $\mathsf{R/W}^{\tau-1} = \mathsf{write}$, then $\mathsf{rData}^{\tau} := \bot$; otherwise $\mathsf{rData}^{\tau} := D[L^{\tau-1}]$. Next it executes the CPU-Step Circuit $C_{\mathsf{CPU}}^{P,\tau}(\mathsf{state}^{\tau}, \mathsf{rData}^{\tau}) = (\mathsf{state}^{\tau+1}, \mathsf{R/W}^{\tau}, L^{\tau}, \mathsf{wData}^{\tau})$. If $\mathsf{R/W}^{\tau} = \mathsf{write}$, then set $D[L^{\tau}] = \mathsf{wData}^{\tau}$. Finally, when $\tau = T$, then $\mathsf{state}^{\tau+1}$ is the output of the program.

3.4 Oblivious RAM

In this subsection, we recall the definition of oblivious RAM [Gol87,Ost90,GO96].

Definition 3 (Oblivious RAM). *An* Oblivious RAM *scheme consists of two procedures* $(\mathsf{OProg}, \mathsf{OData})$ *with the following syntax:*

– $P^* \leftarrow \mathsf{OProg}(1^{\lambda}, 1^{\log M}, 1^T, P)$*: Given a security parameter λ, a memory size M, a program P that runs in time T,* OProg *outputs an probabilistic oblivious program P^* that can access D^* as RAM. A probabilistic RAM program is modeled exactly as a deterministic program except that each step circuit additionally take random coins as input.*

– $D^* \leftarrow \mathsf{OData}(1^{\lambda}, D)$: *Given the security parameter λ, the contents of the database $D \in \{\{0,1\}^N\}^M$, outputs the oblivious database D^*. For convenience, we assume that* OData *works by compiling a program P that writes D to the memory using* OProg *to obtain P^*. It then evaluates the program P^* by using uniform random tape and outputs the contents of the memory as D^*.*

Efficiency. *We require that the run-time of* OData *should be* $M \cdot N \cdot \mathsf{poly}(\log(MN)) \cdot \mathsf{poly}(\lambda)$*, and the run-time of* OProg *should be* $T \cdot \mathsf{poly}(\lambda) \cdot \mathsf{poly}(\log(MN))$*. Finally, the oblivious program P^* itself should run in time* $T' = T \cdot \mathsf{poly}(\lambda) \cdot \mathsf{poly}(\log(MN))$*. Both the new memory size $M' = |D^*|$ and the running time T' should be efficiently computable from M, N, T, and λ.*

Correctness. *Let $P_1, \ldots, P_\ell$ be programs running in polynomial times $t_1, \ldots, t_\ell$ on memory D of size M. Let $x_1, \ldots, x_\ell$ be the inputs and λ be a security parameter. Then we require that:*

$$\Pr[(P_1^*(x_1), \ldots, P_\ell^*(x_\ell))^{D^*} = (P_1(x_1), \ldots, P_\ell(x_\ell))^{D}] = 1$$

where $D^* \leftarrow \mathsf{OData}(1^{\lambda}, D)$, $P_i^* \leftarrow \mathsf{OProg}(1^{\lambda}, 1^{\log M}, 1^T, P_i)$ *and* $(P_1^*(x_1), \ldots, P_\ell^*(x_\ell))^{D^*}$ *indicates running the ORAM programs on D^* sequentially using an uniform random tape.*

Security. *For security, we require that there exists a PPT simulator* Sim *such that for any sequence of programs $P_1, \ldots, P_\ell$ (running in time $t_1, \ldots, t_\ell$ respectively), initial memory data $D \in \{\{0,1\}^N\}^M$, and inputs $x_1, \ldots, x_\ell$ we have that:*

$$\mathsf{MemAccess} \overset{s}{\approx} \mathsf{Sim}(1^{\lambda}, \{1^{t_i}\}_{i=1}^{\ell})$$

where $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$, $D^* \leftarrow \mathsf{OData}(1^\lambda, 1^N, D)$, $P_i^* \leftarrow \mathsf{OProg}(1^\lambda, 1^{\log M}, 1^T, P_i)$ *and* $\mathsf{MemAccess}$ *corresponds to the access pattern of the CPU-step circuits during the sequential execution of the oblivious programs* $(P_1^*(x_1), \ldots, P_\ell^*(x_\ell))^{D^*}$ *using an uniform random tape.*

3.4.1 Strong Localized Randomness For our construction of adaptively secure garbled RAM, we need an additional property called as strong localized randomness property [CCHR16] from an ORAM scheme. We need a slightly stronger formalization than the one given in [CCHR16] (refer to footnote 12).

Strong Localized Randomness. Let $D \in \{\{0,1\}^N\}^M$ be any database and (P, x) be any program/input pair. Let $D^* \leftarrow \mathsf{OData}(1^\lambda, 1^N, D)$ and $P^* \leftarrow \mathsf{OProg}(1^\lambda, 1^{\log M}, 1^T, P)$. Further, let the step circuits of P^* be indicated by $\{C_{\mathsf{CPU}}^{P^*,\tau}\}_{\tau \in [T']}$. Let R be the contents of the random tape used in the execution of P^*.

Definition 4 ([CCHR16]). *We say that an ORAM scheme has strong localized randomness property if there there exists a sequence of efficiently computable values* $\tau_1 < \tau_2 < \ldots < \tau_m$ *where* $\tau_1 = 1$, $\tau_m = T'$ *and* $\tau_t - \tau_{t-1} \leq \mathsf{poly}(\log MN)$ *for all* $t \in [2, m]$ *such that:*

1. *For every* $j \in [m-1]$ *there exists an interval* I_j *(efficiently computable from* j*) of size* $\mathsf{poly}(\log MN, \lambda)$ *s.t. for any* $\tau \in [\tau_j, \tau_{j+1})$*, the random tape accessed by* $C_{\mathsf{CPU}}^{P^*,\tau}$ *is given by* R_{I_j} *(here,* R_{I_j} *denotes the random tape restricted to the interval* I_j*).*
2. *For every* $j, j' \in [m-1]$ *and* $j \neq j'$, $I_j \cap I_{j'} = \emptyset$.
3. *Further, for every* $j \in [m]$*, there exists an* $k < j$ *such that given* $R_{\backslash\{I_k \cup I_j\}}$ *(where* $R_{\backslash\{I_k \cup I_j\}}$ *denotes the content of the random tape except in positions* $I_j \cup I_k$*) and the output of step circuits* $C_{\mathsf{CPU}}^{P^*,\tau}$ *for* $\tau \in [\tau_k, \tau_{k+1})$*, the memory access made by step circuits* $C_{\mathsf{CPU}}^{P^*,\tau}$ *for* $\tau \in [\tau_j, \tau_{j+1})$ *is computationally indistinguishable to random. This* k *is efficiently computable given the program* P *and the input* x*.*[12]

We argue in the full version of our paper that the Chung-Pass ORAM scheme [CP13] where the contents of the database are encrypted using a special encryption scheme satisfies the above definition of strong localized randomness. We now give details on this special encryption scheme. The key generation samples a puncturable PRF key $K \leftarrow \mathsf{PP.KeyGen}(1^\lambda)$. If the τ^{th} step-circuit has to write a value wData to a location L, it first samples $r \leftarrow \{0,1\}^\lambda$ and computes $c = (\tau \| r, \mathsf{PP.Eval}(K, \tau \| r) \oplus \mathsf{wData})$. It writes c to location L. The decryption algorithm uses K to first compute $\mathsf{PP.Eval}(K, \tau \| r)$ and uses it compute wData.

[12] Here, we require that the memory access to be indistinguishable to random even given the outputs of the step circuits $C_{\mathsf{CPU}}^{P^*,\tau}$ for $\tau \in [\tau_k, \tau_{k+1})$. This is where we differ from the definition of [CCHR16].

Remark 2. For the syntax of the ORAM scheme to be consistent with this special encryption scheme, we will use a puncturable PRF to generate the random tape of P^*. This key will also be used implicitly used to derive the key for this special encryption scheme.

3.5 Adaptive Garbled RAM

We now give the definition of adaptive garbled RAM.

Definition 5. *An adaptive garbled RAM scheme* GRAM *consists of the following PPT algorithms satisfying the correctness, efficiency and security properties.*

- $\mathsf{GRAM.Memory}(1^\lambda, D)$*: It is a PPT algorithm that takes the security parameter* 1^λ *and a database* $D \in \{0,1\}^M$ *as input and outputs a garbled database* $\widetilde{D}$ *and a secret key* SK*.*
- $\mathsf{GRAM.Program}(SK, i, P)$*: It is a PPT algorithm that takes as input a secret key* SK*, a sequence number* i*, and a program* P *as input (represented as a sequence of CPU steps) and outputs a garbled program* $\widetilde{P}$*.*
- $\mathsf{GRAM.Input}(SK, i, x)$*: It is a PPT algorithm that takes as input a secret key* SK*, a sequence number* i *and a string* x *as input and outputs the garbled input* $\widetilde{x}$*.*
- $\mathsf{GRAM.Eval}^{\widetilde{D}}(\mathsf{st}, \widetilde{P}, \widetilde{x})$*: It is a RAM program with random read write access to* $\widetilde{D}$*. It takes the state information* st*, garbled program* $\widetilde{P}$ *and the garbled input* $\widetilde{x}$ *as input and outputs a string* y *and updated database* $\widetilde{D}'$*.*

Correctness. *We say that a garbled RAM* GRAM *is correct if for every database* D*,* $t = \mathsf{poly}(\lambda)$ *and every sequence of program and input pair* $\{(P_1, x_1), \ldots, (P_t, x_t)\}$ *we have that*

$$\Pr[\mathsf{Expt}_{\mathsf{correctness}}(1^\lambda, \mathsf{UGRAM}) = 1] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{Expt}_{\mathsf{correctness}}$ *is defined in Figure 5.*

Adaptive Security. *We say that* GRAM *satisfies adaptive security if there exists (stateful) simulators* $(\mathsf{SimD}, \mathsf{SimP}, \mathsf{SimIn})$ *such that for all* t *that is polynomial in the security parameter* λ *and for all polynomial time (stateful) adversaries* $\mathcal{A}$ *, we have that*

$$\left|\Pr[\mathsf{Expt}_{\mathsf{real}}(1^\lambda, \mathsf{GRAM}, \mathcal{A}) = 1] - \Pr[\mathsf{Expt}_{\mathsf{ideal}}(1^\lambda, \mathsf{Sim}, \mathcal{A}) = 1]\right| \leq \mathsf{negl}$$

where $\mathsf{Expt}_{\mathsf{real}}, \mathsf{Expt}_{\mathsf{ideal}}$ *are defined in Figure 4.*

Efficiency. *We require the following efficiency properties from a* UGRAM *scheme.*

- *The running time of* $\mathsf{GRAM.Memory}$ *should be bounded by* $M \cdot \mathsf{poly}(\log M) \cdot \mathsf{poly}(\lambda)$*.*

```
– $(\widetilde{D}, SK) \leftarrow \mathsf{GRAM.Memory}(1^\lambda, D)$.
– Set $D_1 := D$, $\widetilde{D}_1 := \widetilde{D}$ and $\mathsf{st} = \perp$.
– for every $i$ from 1 to $t$
    • $\widetilde{P}_i \leftarrow \mathsf{GRAM.Program}(SK, i, P_i)$ .
    • $\widetilde{x}_i \leftarrow \mathsf{GRAM.Input}(SK, i, x_i)$.
    • Compute $(y_i, D_{i+1}) := P_i^{D_i}(x_i)$ and $(\widetilde{y}_i, \widetilde{D}_{i+1}, \mathsf{st}) := \mathsf{UGRAM.Eval}^{\widetilde{D}_i}(i, \mathsf{st}, \widetilde{P}_i, \widetilde{x}_i)$.
– Output 1 if there exists an $i \in [t]$ such that $\widetilde{y}_i \neq y_i$.
```

Figure 3: Correctness Experiment for GRAM

```
$\mathsf{Expt}_{\mathsf{real}}[1^\lambda, \mathsf{GRAM}, \mathcal{A}]$

– $D \leftarrow \mathcal{A}(1^\lambda)$ where $D \in \{0,1\}^M$
– $(\widetilde{D}, SK) \leftarrow \mathsf{GRAM.Memory}(1^\lambda, D)$.
– for every $i$ from 1 to $t$
    • $P_i \leftarrow \mathcal{A}(\widetilde{D}, \{(\widetilde{P}_1, \widetilde{x}_1), \ldots, (\widetilde{P}_{i-1}, \widetilde{x}_{i-1})\})$.
    • $\widetilde{P}_i \leftarrow \mathsf{GRAM.Program}(SK, i, P_i)$.
    • $x_i \leftarrow \mathcal{A}(\widetilde{D}, \{(\widetilde{P}_1, \widetilde{x}_1), \ldots, (\widetilde{P}_{i-1}, \widetilde{x}_{i-1})\}, \widetilde{P}_i)$.
    • $\widetilde{x}_i \leftarrow \mathsf{GRAM.Input}(SK, i, x_i)$.
– Output $\mathcal{A}(\{(\widetilde{P}_1, \widetilde{x}_1), \ldots, (\widetilde{P}_t, \widetilde{x}_t)\})$.
```

```
$\mathsf{Expt}_{\mathsf{ideal}}[1^\lambda, \mathsf{Sim}, \mathcal{A}]$

– $D \leftarrow \mathcal{A}(1^\lambda)$ where $D \in \{0,1\}^M$.
– $(\widetilde{D}, \mathsf{st}) \leftarrow \mathsf{SimD}(1^\lambda, 1^M)$.
– for every $i$ from 1 to $t$
    • $P_i \leftarrow \mathcal{A}(\widetilde{D}, \{(\widetilde{P}_1, \widetilde{x}_1), \ldots, (\widetilde{P}_{i-1}, \widetilde{x}_{i-1})\})$.
    • $(\widetilde{P}_i, \mathsf{st}) \leftarrow \mathsf{SimP}(1^{|P_i|}, \mathsf{st})$ .
    • $x_i \leftarrow \mathcal{A}(\widetilde{D}, \{(\widetilde{P}_1, \widetilde{x}_1), \ldots, (\widetilde{P}_{i-1}, \widetilde{x}_{i-1})\}, \widetilde{P}_i)$.
    • $(y_i, D_{i+1}) := P_i^{D_i}(x_i)$ where $D_1 := D$.
    • $\widetilde{x}_i \leftarrow \mathsf{SimIn}(\mathsf{st}, y_i)$.
– Output $\mathcal{A}(\{(\widetilde{P}_1, \widetilde{x}_1), \ldots, (\widetilde{P}_t, \widetilde{x}_t)\})$.
```

Figure 4: Adaptive Security Experiment for GRAM

– *The running time of* GRAM.Program *should be bounded by* $T \cdot \mathsf{poly}(\log M) \cdot \mathsf{poly}(\lambda)$ *where* T *is the number of CPU steps in the description of the program* P.
– *The running time of* GRAM.Input *should be bounded by* $|x| \cdot \mathsf{poly}(\log M, \log T) \cdot \mathsf{poly}(\lambda)$.
– *The running time of* GRAM.Eval *should be bounded by* $T \cdot \mathsf{poly}(\log M) \cdot \mathsf{poly}(\lambda)$ *where* T *is the number of CPU steps in the description of the program* P.

4 Adaptive Garbled RAM with Unprotected Memory Access

Towards our goal of constructing an adaptive garbled RAM, we first construct an intermediate primitive with weaker security guarantees. We call this primitive as *adaptive garbled RAM with unprotected memory access.* Informally, a garbled RAM scheme has unprotected memory access if both the contents of the database and the access to the database are revealed in the clear to the adversary. We differ from the security definition given in [GHL+14] in three aspects. Firstly, we give an indistinguishability style definition for security whereas [GHL+14] give a simulation style definition. The indistinguishability based definition makes it easier to get full-fledged adaptive security later. Secondly and most importantly, we allow the adversary to adaptively choose the inputs based on the garbled program. Thirdly, we also require the garbled RAM scheme to satisfy a special property called as *equivocability.* Informally, equivocability requires that the real garbling of a program P is indistinguishable to a simulated garbling where the simulator is not provided with the description of the step circuits for a certain number of time steps (this number is given by the equivocation parameter). Later, when the input is specified, the simulator is given the output of these step circuits and must come-up with an appropriate garbled input.

We now give the formal definition of this primitive.

Definition 6. *An adaptive garbled RAM scheme with unprotected memory access* UGRAM *consists of the following PPT algorithms satisfying the correctness, efficiency and security properties.*

- $\mathsf{UGRAM.Memory}(1^\lambda, 1^n, D)$*: It is a PPT algorithm that takes the security parameter* 1^λ*, an equivocation parameter* n *and a database* $D \in \{\{0,1\}^N\}^M$ *as input and outputs a garbled database* $\widetilde{D}$ *and a secret key* SK*.*
- $\mathsf{UGRAM.Program}(SK, i, P)$*: It is a PPT algorithm that takes as input a secret key* SK*, a sequence number* i*, and a program* P *as input (represented as a sequence of CPU steps) and outputs a garbled program* $\widetilde{P}$*.*
- $\mathsf{UGRAM.Input}(SK, i, x)$*: It is a PPT algorithm that takes as input a secret key* SK*, a sequence number* i *and a string* x *as input and outputs the garbled input* $\widetilde{x}$*.*
- $\mathsf{UGRAM.Eval}^{\widetilde{D}}(\mathsf{st}, \widetilde{P}, \widetilde{x})$*: It is a RAM program with random read write access to* $\widetilde{D}$*. It takes the state information* st*, garbled program* $\widetilde{P}$ *and the garbled input* $\widetilde{x}$ *as input and outputs a string* y *and updated database* $\widetilde{D}'$*.*

Correctness. *We say that a garbled RAM* UGRAM *is correct if for every database* D*,* $t = \mathsf{poly}(\lambda)$ *and every sequence of program and input pair* $\{(P_1, x_1), \ldots, (P_t, x_t)\}$ *we have that*

$$\Pr[\mathsf{Expt}_{\mathsf{correctness}}(1^\lambda, \mathsf{UGRAM}) = 1] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{Expt}_{\mathsf{correctness}}$ *is defined in Figure 5.*

Security. *We require the following two properties to hold.*

- ***Equivocability.*** *There exists a simulator* Sim *such that for any non-uniform PPT stateful adversary* $\mathcal{A}$ *and* $t = \mathsf{poly}(\lambda)$ *we require that:*

$$\left|\Pr[\mathsf{Expt}_{\mathsf{equiv}}(1^\lambda, \mathcal{A}, 0) = 1] - \Pr[\mathsf{Expt}_{\mathsf{equiv}}(1^\lambda, \mathcal{A}, 1) = 1]\right| \leq \mathsf{negl}(\lambda)$$

```
– $(\widetilde{D}, SK) \leftarrow \mathsf{UGRAM.Memory}(1^\lambda, 1^n, D)$.
– Set $D_1 := D$, $\widetilde{D}_1 := \widetilde{D}$ and $\mathsf{st} = \perp$.
– for every $i$ from 1 to $t$
    • $\widetilde{P}_i \leftarrow \mathsf{UGRAM.Program}(SK, i, P_i)$ .
    • $\widetilde{x}_i \leftarrow \mathsf{UGRAM.Input}(SK, i, x_i)$.
    • Compute $(y_i, D_{i+1}) := P_i^{D_i}(x_i)$ and $(\widetilde{y}_i, \widetilde{D}_{i+1}, \mathsf{st}) := \mathsf{UGRAM.Eval}^{\widetilde{D}_i}(i, \mathsf{st}, \widetilde{P}_i, \widetilde{x}_i)$.
– Output 1 if there exists an $i \in [t]$ such that $\widetilde{y}_i \neq y_i$.
```

Figure 5: Correctness Experiment for UGRAM

where $\mathsf{Expt}_{\mathsf{equiv}}(1^\lambda, \mathcal{A}, b)$ *is described in Figure 6.*

– ***Adaptive Security.*** *For any non-uniform PPT stateful adversary* $\mathcal{A}$ *and* $t = \mathsf{poly}(\lambda)$ *we require that:*

$$\left|\Pr[\mathsf{Expt}_{\mathsf{UGRAM}}(1^\lambda, \mathcal{A}, 0) = 1] - \Pr[\mathsf{Expt}_{\mathsf{UGRAM}}(1^\lambda, \mathcal{A}, 1) = 1]\right| \leq \mathsf{negl}(\lambda)$$

where $\mathsf{Expt}_{\mathsf{UGRAM}}(1^\lambda, \mathcal{A}, b)$ *is described in Figure 7.*

```
1. $D \leftarrow \mathcal{A}(1^\lambda, 1^n)$.
2. $\widetilde{D}$ is computed as follows:
   (a) If $b = 0$ : $(\widetilde{D}, SK) \leftarrow \mathsf{UGRAM.Memory}(1^\lambda, 1^n, D)$.
   (b) If $b = 1$ : $\widetilde{D} \leftarrow \mathsf{Sim}(1^\lambda, 1^n, D)$.
3. for each $i$ from $t$:
   (a) $(P_i, I) \leftarrow \mathcal{A}(\widetilde{D}, \{\widetilde{P}_j, \widetilde{x}_j\}_{j \in [i-1]})$ where $I \subset [|P_i|]$ and $|I| \leq n$.
   (b) $\widetilde{P}_i$ is computed as follows:
       i. If $b = 0$ : $\widetilde{P}_i \leftarrow \mathsf{UGRAM.Program}(SK, i, P_i)$.
       ii. If $b = 1$ : $\widetilde{P}_i \leftarrow \mathsf{Sim}(\{C_{\mathsf{CPU}}^{P_i, t}\}_{t \notin I})$
   (c) $x_i \leftarrow \mathcal{A}(\{\widetilde{P}_j, \widetilde{x}_j\}_{j \in [i-1]}, \widetilde{P}_i)$.
   (d) $\widetilde{x}_i$ is computed as follows:
       i. If $b = 0$ : $\widetilde{x}_i \leftarrow \mathsf{UGRAM.Input}(SK, i, x_i)$
       ii. If $b = 1$ : $\widetilde{x}_i \leftarrow \mathsf{Sim}(x_i, \{y_t\}_{t \in I})$ where $y_t$ is the o/p of $C_{\mathsf{CPU}}^{P_i, t}$ when $P_i$ is executed with $x_i$.
4. $b' \leftarrow \mathcal{A}(\{\widetilde{P}_j, \widetilde{x}_j\}_{j \in [t]})$.
5. Output $b'$.
```

Figure 6: $\mathsf{Expt}_{\mathsf{equiv}}(1^\lambda, \mathcal{A}, b)$

```
1. $D \leftarrow \mathcal{A}(1^\lambda, 1^n)$.
2. $(\widetilde{D}, SK) \leftarrow \mathsf{UGRAM.Memory}(1^\lambda, 1^n, D)$.
3. for x each $i$ from $t$:
   (a) $(P_{i,0}, P_{i,1}) \leftarrow \mathcal{A}(\widetilde{D}, \{\widetilde{P}_j, \widetilde{x}_j\}_{j\in[i-1]})$.
   (b) $\widetilde{P}_i$ is computed as follows:
       i. If $b = 0 : \widetilde{P}_i \leftarrow \mathsf{UGRAM.Program}(SK, i, P_{i,0})$.
       ii. If $b = 1 : \widetilde{P}_i \leftarrow \mathsf{UGRAM.Program}(SK, i, P_{i,1})$
   (c) $x_i \leftarrow \mathcal{A}(\{\widetilde{P}_j, \widetilde{x}_j\}_{j\in[i-1]}, \widetilde{P}_i)$.
   (d) $\widetilde{x}_i \leftarrow \mathsf{UGRAM.Input}(i, SK, x_i)$
4. $b' \leftarrow \mathcal{A}(\{\widetilde{P}_j, \widetilde{x}_j\}_{j\in[t]})$.
5. Output $b'$ if the output of each step circuit in $P^D_{i,0}(x_i)$ is same as $P^D_{i,1}(x_i)$ for
   every $i \in [t]$.
```

Figure 7: $\mathsf{Expt}_{\mathsf{UGRAM}}(1^\lambda, \mathcal{A}, b)$

Efficiency. *We require the following efficiency properties from a* UGRAM *scheme.*

- *The running time of* UGRAM.Memory *should be bounded by* $MN \cdot \mathsf{poly}(\log MN) \cdot \mathsf{poly}(\lambda)$.
- *The running time of* UGRAM.Program *should be bounded by* $T \cdot \mathsf{poly}(\log MN) \cdot \mathsf{poly}(\lambda)$ *where* T *is the number of CPU steps in the description of the program* P.
- *The running time of* UGRAM.Input *should be bounded by* $n \cdot |x| \cdot \mathsf{poly}(\log MN, \log T) \cdot \mathsf{poly}(\lambda)$.
- *The running time of* UGRAM.Eval *should be bounded by* $T \cdot \mathsf{poly}(\log MN, \log T) \cdot \mathsf{poly}(\lambda)$ *where* T *is the number of CPU steps in the description of the program* P.

4.1 Construction

In this subsection, we give a construction of adaptive garbled RAM with unprotected memory access from updatable laconic oblivious transfer, somewhere equivocal encryption and garbling scheme for circuits with selective security using the techniques developed in the construction of adaptive garbled circuits [GS18a]. Our main theorem is:

Theorem 4. *Assuming the existence of updatable laconic oblivious transfer, somewhere equivocal encryption, a pseudorandom function and garbling scheme for circuits with selective security, there exists a construction of adaptive garbled RAM with unprotected memory access.*

Construction. We give the formal description of the construction in Figure 8.

We use a somewhere equivocal encryption with block length set to $|\widetilde{\mathsf{SC}}_\tau|$ where $\widetilde{\mathsf{SC}}_\tau$ denotes the garbled version of the step circuit SC described in Figure 9, the message length to be T (which is the running time of the program P) and the equivocation parameter to be $t + \log T$ where t is the actual equivocation parameter for the UGRAM scheme.

Correctness. The correctness of the above construction follows from a simple inductive argument that for each step $\tau \in [|P|]$, the state and the database are updated correctly at the end of the execution of $\widetilde{\mathsf{SC}}_\tau$. The base case is $\tau = 0$. In order to prove the inductive step for a step τ, observe that if the step τ outputs a read then labels recovered in Step 4.(c).(ii) of $\mathsf{AdpEvalCkt}$ correspond to data block in the location requested. Otherwise, the labels recovered in Step 4(b).(ii) of $\mathsf{AdpEvalCkt}$ corresponds to the updated value of the digest with the corresponding block written to the database.

Efficiency. The efficiency of our construction directly follows from the efficiency of updatable laconic oblivious transfer and the parameters set for the somewhere equivocal encryption. In particular, the running time of $\mathsf{UGRAM.Memory}$ is $D \cdot \mathsf{poly}(\lambda)$, $\mathsf{UGRAM.Program}$ is $T \cdot \mathsf{poly}(\log MN, \lambda)$ and that of $\mathsf{UGRAM.Input}$ is $n|x| \cdot \mathsf{poly}(\log M, \log T, \lambda)$. The running time of $\mathsf{UGRAM.Eval}$ is $T \cdot \mathsf{poly}(\log M, \log T, \lambda)$.

Security. We prove the security of this construction in the full version of our paper.

5 Timed Encryption

In this section, we give the definition and construction of a timed encryption scheme. We will use a timed encryption scheme in the construction of adaptive garbled RAM in the next section.

A timed encryption scheme is a symmetric key encryption scheme with some special properties. In this encryption scheme, every message is encrypted with respect to a timestamp time. Additionally, there is a special algorithm called as constrain that takes an encryption key K and a timestamp time' as input and outputs a time constrained key $K[\mathsf{time}']$. A time constrained key $K[\mathsf{time}']$ can be used to decrypt any ciphertext that is encrypted with respect to timestamp $\mathsf{time} < \mathsf{time}'$. For security, we require that knowledge of a time constrained key does not help an adversary to distinguish between encryptions of two messages that are encrypted with respect to some future timestamp.

Definition 7. *A timed encryption scheme is a tuple of algorithms* ($\mathsf{TE.KeyGen}$, $\mathsf{TE.Enc}$, $\mathsf{TE.Dec}$, $\mathsf{TE.Constrain}$) *with the following syntax.*

- $\mathsf{TE.KeyGen}(1^\lambda)$: *It is a randomized algorithm that takes the security parameter* 1^λ *and outputs a key* K.
- $\mathsf{TE.Constrain}(K, \mathsf{time})$: *It is a deterministic algorithm that takes a key* K *and a timestamp* $\mathsf{time} \in [0, 2^\lambda - 1]$ *and outputs a time-constrained key* $K[\mathsf{time}]$.

```
$\mathsf{UGRAM.Memory}(1^\lambda, 1^t, D)$: On input a database $D \in \{\{0,1\}^N\}^M$ do:
    1. Sample $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda, 1^N)$ and $K \leftarrow \mathsf{PRFKeyGen}(1^\lambda)$ defining $\mathsf{PRF}_K : \{0,1\}^{2\lambda+1} \to \{0,1\}^\lambda$.
    2. For each $k \in [\lambda]$ and $b \in \{0,1\}$, compute $\mathsf{lab}^1_{k,b} := \mathsf{PRF}_K(1\|k\|b)$.
    3. Compute $(\mathsf{d}, \widehat{D}) = \mathsf{Hash}(\mathsf{crs}, D)$.
    4. Output $\widehat{D}, \{\mathsf{lab}^1_{k,\mathsf{d}_k}\}_{k\in[\lambda]}$ as the garbled memory and $(K, \mathsf{crs})$ as the secret key.
$\mathsf{UGRAM.Program}(SK, i, P)$: On input $SK = (K, \mathsf{crs})$, sequence number $i$, and a program $P$ (with $T$ step-circuits) do:
    1. For each step $\tau \in [2, T]$, $k \in [\lambda + n + N]$ and $b \in \{0,1\}$,
        (a) Sample $\mathsf{lab}^\tau_{k,b} \leftarrow \{0,1\}^\lambda$.
        (b) Set $\mathsf{lab}^1_{k,b} := \mathsf{PRF}_K(i\|k\|b)$ and $\mathsf{lab}^{T+1}_{k,b} := \mathsf{PRF}_K((i+1)\|k\|b)$.
        We use $\{\mathsf{lab}^\tau_{k,b}\}$ to denote $\{\mathsf{lab}^\tau_{k,b}\}_{k\in[\lambda+n+N], b\in\{0,1\}}$.
    2. for each $\tau$ from $T$ down to 1 do:
        (a) Compute $\widetilde{\mathsf{SC}}_\tau \leftarrow \mathsf{GarbleCkt}\left(1^\lambda, \mathsf{SC}[\mathsf{crs}, \tau, \{\mathsf{lab}^{\tau+1}_{k,b}\}], \{\mathsf{lab}^\tau_{k,b}\}\right)$ where the step-circuit $\mathsf{SC}$ is described in Figure 9.
    3. Compute $\mathsf{key} = \mathsf{KeyGen}(1^\lambda; \mathsf{PRF}_K(i\|0^\lambda\|0))$
    4. Compute $c \leftarrow \mathsf{Enc}(\mathsf{key}, \{\widetilde{\mathsf{SC}}_\tau\}_{\tau\in[T]})$ and output $\widetilde{P} := c$.
$\mathsf{UGRAM.Input}(SK, i, x)$ : On input the secret key $SK = (K, \mathsf{crs})$, sequence number $i$ and a string $x \in \{0,1\}^n$ do:
    1. For each $k \in [\lambda + n + N]$ and $b \in \{0,1\}$, compute $\mathsf{lab}^1_{k,b} := \mathsf{PRF}_K(i\|k\|b)$.
    2. Compute $\mathsf{key} = \mathsf{KeyGen}(1^\lambda; \mathsf{PRF}_K(i\|0^\lambda\|0))$.
    3. Output $\widetilde{x} := \left(\mathsf{key}, \{\mathsf{lab}^1_{k,x_k}\}_{k\in[\lambda+1,\lambda+n]}, \{\mathsf{lab}^1_{k,0}\}_{k\in[n+\lambda+1,n+\lambda+N]}\right)$.
$\mathsf{UGRAM.Eval}^{\widehat{D}}(i, \mathsf{st}, \widetilde{P}, \widetilde{x})$ : On input $i$, state $\mathsf{st}$, the garbled program $\widetilde{P}$, and garbled input $\widetilde{x}$ do:
    1. Parse $\widetilde{x}$ as $\left(\mathsf{key}, \{\mathsf{lab}_k\}_{k\in[\lambda+1,n+\lambda+N]}\right)$ and $\widetilde{P}$ as $c$.
    2. If $i = 1$, obtain $\{\mathsf{lab}_k\}_{k\in[\lambda]}$ from garbled memory; else, parse $\mathsf{st}$ as $\{\mathsf{lab}_k\}_{k\in[\lambda]}$.
    3. Compute $\{\widetilde{\mathsf{SC}}_\tau\}_{\tau\in[T]} := \mathsf{Dec}(\mathsf{key}, c)$ and set $\overline{\mathsf{lab}} := \{\mathsf{lab}_k\}_{k\in[n+\lambda+N]}$.
    4. for each $\tau$ from 1 to $T$ do:
        (a) Compute $(\mathsf{R/W}, L, A, \{\mathsf{lab}_k\}_{k\in[\lambda+1,\lambda+n]}, B) := \mathsf{EvalCkt}(\widetilde{\mathsf{SC}}_\tau, \overline{\mathsf{lab}})$.
        (b) If $\mathsf{R/W} = \mathsf{write}$,
            i. Parse $A$ as $(e_w, \mathsf{wData})$ and $B$ as $\{\mathsf{lab}_k\}_{k\in[\lambda+1,n+\lambda+N]}$.
            ii. $\{\mathsf{lab}_k\}_{k\in[\lambda]} \leftarrow \mathsf{ReceiveWrite}^{\widehat{D}}(\mathsf{crs}, L, \mathsf{wData}, e_w)$
        (c) else,
            i. Parse $A$ as $\{\mathsf{lab}_k\}_{k\in[n+\lambda]}$ and $B$ as $e$.
            ii. $\{\mathsf{lab}_k\}_{k\in[n+\lambda+1,n+\lambda+N]} \leftarrow \mathsf{Receive}^{\widehat{D}}(\mathsf{crs}, L, e)$
        (d) Set $\overline{\mathsf{lab}} := \{\mathsf{lab}_k\}_{k\in[n+\lambda+N]}$.
    5. Parse $\overline{\mathsf{lab}}$ as $\{\mathsf{lab}_k\}_{k\in[n+\lambda+N]}$. Output $\{\mathsf{lab}_k\}_{k\in[\lambda+1,n+\lambda]}$ and $\mathsf{st} := \{\mathsf{lab}_k\}_{k\in[\lambda]}$.
```

Figure 8: Adaptive Garbled RAM with Unprotected Memory Access

Step Circuit SC

Input: A digest d, state state and a block rData.
Hardcoded: The common reference string crs, the step number τ and a set of labels $\{\mathsf{lab}_{k,b}\}$.

```
1. Compute $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) := C_{\mathsf{CPU}}^{P,\tau}(\mathsf{state}, \mathsf{rData})$.
2. If $\tau = T$, reset $\mathsf{lab}_{k,b} = b$ for all $k \in [\lambda+1, \lambda+n]$ and $b \in \{0,1\}$.
3. if $\mathsf{R/W} = \mathsf{write}$ do:
    (a) Compute $e_w \leftarrow \mathsf{SendWrite}(\mathsf{crs}, \mathsf{d}, L, \mathsf{wData}, \{\mathsf{lab}_{k,b}\}_{k\in[\lambda], b\in\{0,1\}})$.
    (b) Output $(\mathsf{R/W}, L, e_w, \mathsf{wData}, \{\mathsf{lab}_{k,\mathsf{state}'_{k-\lambda}}\}_{k\in[\lambda+1,\lambda+n]}, \{\mathsf{lab}_{k,0}\}_{k\in[n+\lambda+1, n+\lambda+N]})$.
4. else,
    (a) Compute $e \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{d}, L, \{\mathsf{lab}_{k,b}\}_{k\in[n+\lambda+1,n+\lambda+N], b\in\{0,1\}})$.
    (b) $(\mathsf{R/W}, L, \{\mathsf{lab}_{k,\mathsf{d}_k}\}_{k\in[\lambda]}, \{\mathsf{lab}_{k,\mathsf{state}'_{k-\lambda}}\}_{k\in[\lambda+1,\lambda+n]}, e)$.
```

Figure 9: Description of the Step Circuit

- $\mathsf{TE.Enc}(K, \mathsf{time}, m)$: *It is a randomized algorithm that takes a key K, a timestamp* time *and a message m as input and outputs a ciphertext c or $\perp$.*
- $\mathsf{TE.Dec}(K, c)$: *It is a deterministic algorithm that takes a key K and a ciphertext c as input and outputs a message m.*

We require a timed encryption scheme to follow the following properties.

Correctness. *We require that for all messages m and for all timestamps* $\mathsf{time}_1 \leq \mathsf{time}_2$*:*

$$\Pr[\mathsf{TE.Dec}(K[\mathsf{time}_2], c) = m] = 1$$

where $K \leftarrow \mathsf{TE.KeyGen}(1^\lambda)$, $K[\mathsf{time}_2] := \mathsf{TE.Constrain}(K, \mathsf{time}_2)$ *and* $c \leftarrow \mathsf{TE.Enc}(K, \mathsf{time}_1, m)$.

Encrypting with Constrained Key. *For any message m and timestamps* $\mathsf{time}_1 \leq \mathsf{time}_2$*, we require that:*

$$\{\mathsf{TE.Enc}(K, \mathsf{time}_1, m)\} \approx \{\mathsf{TE.Enc}(K[\mathsf{time}_2], \mathsf{time}_1, m)\}$$

where $K \leftarrow \mathsf{TE.KeyGen}(1^\lambda)$, $K[\mathsf{time}_2] := \mathsf{TE.Constrain}(K, \mathsf{time}_2)$ *and $\approx$ denotes that the two distributions are identical.*

Security. *For any two messages m_0, m_1 and timestamps* $(\mathsf{time}, \{\mathsf{time}_i\}_{i\in[t]})$ *where* $\mathsf{time}_i < \mathsf{time}$ *for all $i \in [t]$, we require that:*

$$\{\{K[\mathsf{time}_i]\}_{i\in[t]}, \mathsf{TE.Enc}(K, \mathsf{time}, m_0)\} \overset{c}{\approx} \{\{K[\mathsf{time}_i]\}_{i\in[t]}, \mathsf{TE.Enc}(K, \mathsf{time}, m_1)\}$$

where $K \leftarrow \mathsf{TE.KeyGen}(1^\lambda)$ *and* $K[\mathsf{time}_i] := \mathsf{TE.Constrain}(K, \mathsf{time}_i)$ *for every $i \in [t]$.*

We prove the following theorem in the full version of our paper.

Theorem 5. *Assuming the existence of one-way functions, there exists a construction of timed encryption.*

6 Construction of Adaptive Garbled RAM

In this section, we give a construction of adaptive garbled RAM. We make use of the following primitives.

- A timed encryption scheme $(\mathsf{TE.KeyGen}, \mathsf{TE.Enc}, \mathsf{TE.Dec}, \mathsf{TE.Constrain})$. Let N be the output length of $\mathsf{TE.Enc}$ when encrypting single bit messages.
- A puncturable pseudorandom function $(\mathsf{PP.KeyGen}, \mathsf{PP.Eval}, \mathsf{PP.Punc})$.
- An oblivious RAM scheme $(\mathsf{OData}, \mathsf{OProg})$ with strong localized randomness.
- An adaptive garbled RAM scheme UGRAM with unprotected memory access.

The formal description of our construction appears in Figure 10.

Correctness. We give an informal argument for correctness. The only difference between UGRAM and the construction we give in Figure 10 is that we encrypt the database using a timed encryption scheme and encode it using a ORAM scheme. To argue the correctness of our construction, it is sufficient to argue that each step circuit SC faithfully emulates the corresponding step circuit of P^*. Let $\mathsf{SC}^{i,\tau}$ be the step circuit that corresponds to the τ^{th} step of the i^{th} program P_i. We observe that any point in time the L^{th} location of the database $\widehat{D}$ is an encryption of the actual data bit with respect to timestamp $\mathsf{time} := (i' \| \tau')$ where $\mathsf{SC}^{i',\tau'}$ last wrote at the L^{th} location. It now follows from this invariant and the correctness of the timed encryption scheme that the hardwired constrained key $K[i\|\tau]$ in $\mathsf{SC}^{i,\tau}$ can be used to decrypt the read block X as the step that last modified this block has a timestamp that is less than $(i\|\tau)$.

Efficiency. We note that setting the equivocation parameter $n = \mathsf{poly}(\log MN)$, we obtain that the running time of $\mathsf{GRAM.Input}$ is $|x| \cdot \mathsf{poly}(\lambda, \log MN)$. The rest of the efficiency criterion follow directly from the efficiency of adaptive garbled RAM with unprotected memory access.
Security. We give the proof of security in the full version of our paper.

References

[ACC$^+$16] Prabhanjan Ananth, Yu-Chi Chen, Kai-Min Chung, Huijia Lin, and Wei-Kai Lin. Delegating RAM computations with adaptive soundness and privacy. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 3–30. Springer, Heidelberg, October / November 2016.

[AIK04] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. In *45th FOCS*, pages 166–175. IEEE Computer Society Press, October 2004.

```
GRAM.Memory$(1^\lambda, D)$: On input the database $D \in \{0,1\}^M$:
    1. Sample $K \leftarrow \mathsf{TE.KeyGen}(1^\lambda)$ and $S \leftarrow \mathsf{PRFKeyGen}(1^\lambda)$ defining $\mathsf{PRF}_S : \{0,1\}^\lambda \to \{0,1\}^n$ (where $n$ is the input length of each program).
    2. Initialize an empty array $\widehat{D}$ of $M$ blocks with block length $N$.
    3. for each $i$ from 1 to $M$ do:
        (a) Set $\widehat{D}[i] \leftarrow \mathsf{TE.Enc}(K, 0^\lambda, D[i])$.
    4. $D^* \leftarrow \mathsf{OData}(1^\lambda, 1^N, \widehat{D})$.
    5. $(\widetilde{D}, SK) \leftarrow \mathsf{UGRAM.Memory}(1^\lambda, 1^t, D^*)$ where $t = \mathsf{poly}(\log MN)$.
    6. Output $\widetilde{D}$ as the garbled memory and $(K, S, SK)$ as the secret key.
GRAM.Program$(SK', i, P)$: On input $SK' = (K, S, SK)$, sequence number $i$, and a program $P$:
    1. Sample $K' \leftarrow \mathsf{PP.KeyGen}(1^\lambda)$
    2. $P^* \leftarrow \mathsf{OProg}(1^\lambda, 1^{\log M}, 1^T, P)$ where $P^*$ runs in time $T'$.
    3. For each $\tau \in [T']$, compute $K[(i\|\tau)] \leftarrow \mathsf{TE.Constrain}(K, (i\|\tau))$ where $(i\|\tau)$ is expressed as a $\lambda$-bit string.
    4. Compute $r := \mathsf{PRF}_S(i)$.
    5. Let $\tau_1, \ldots, \tau_m$ be the sequence of values guaranteed by strong localized randomness.
    6. for each $\tau \in [T']$ do:
        (a) Let $j \in [m-1]$ be such that $\tau \in [\tau_j, \tau_{j+1})$.
        (b) Let $C^\tau_{\mathsf{CPU}} := \mathsf{SC}_\tau[i, \tau, K[(i\|\tau)], I_j, K', r']$ where $r' = r$ if $\tau = T'$, else $r' = \perp$. The step circuit $\mathsf{SC}$ is described in Figure 11
    7. Construct a RAM program $P'$ with step-circuits given by $\{C^\tau_{\mathsf{CPU}}\}$.
    8. $\widetilde{P} \leftarrow \mathsf{UGRAM.Program}(SK, i, P')$.
    9. Output $\widetilde{P}$.
GRAM.Input$(SK', i, P)$: On input $SK' = (K, S, SK)$, $i$ and $x$:
    1. Compute $r = \mathsf{PRF}_S(i)$
    2. Compute $\widehat{x} \leftarrow \mathsf{UGRAM.Input}(SK, i, x)$
    3. Output $\widetilde{x} = (\widehat{x}, r)$.
GRAM.Eval$^{\widetilde{D}}(i, \mathsf{st}, \widetilde{P}, \widetilde{x})$: On input state $\mathsf{st}$, the garbled program $\widetilde{P}$, and garbled input $\widetilde{x}$:
    1. Compute $(y, \mathsf{st}') \leftarrow \mathsf{UGRAM.Eval}^{\widetilde{D}}(\mathsf{st}, \widetilde{P}, \widehat{x})$ and update $\mathsf{st}$ to $\mathsf{st}'$. Output $y \oplus r$.
```

Figure 10: Construction of Adaptive GRAM

```
Step Circuit $\mathsf{SC}_\tau$

Input: A ciphertext $c_{\mathsf{CPU}}$ and a data block $X \in \{0,1\}^N$.
Hardcoded: The sequence number $i$, step number $\tau$, the constrained key $K[(i\|\tau)]$,
the interval $I_j$, the key $K'$ and a string $r'$.

1. Compute $\mathsf{rData} := \mathsf{TE.Dec}(K[(i\|\tau)], X)$ and $\mathsf{state} = \mathsf{TE.Dec}(K[(i\|\tau)], c_{\mathsf{CPU}})$.
2. Compute $R_{I_j} = \mathsf{PP.Eval}(K', I_j)$.
3. Compute $(\mathsf{R/W}, L, \mathsf{state}', \mathsf{wData}) := C_{\mathsf{CPU}}^{P^*,\tau}(\mathsf{state}, \mathsf{rData}, R_{I_j})$.
4. if $\tau = T'$, then output $c'_{\mathsf{CPU}} = \mathsf{state}' \oplus r'$; else $c'_{\mathsf{CPU}} = \mathsf{TE.Enc}(K[(i\|\tau)], \mathsf{state}')$.
5. else if $\mathsf{R/W} = \mathsf{write}$ do:
   (a) Compute $X' \leftarrow \mathsf{TE.Enc}(K[i\|\tau], (i,\tau), \mathsf{wData})$.
   (b) Output $(c'_{\mathsf{CPU}}, \mathsf{R/W}, L, X')$.
6. else if $\mathsf{R/W} = \mathsf{read}$, output $(c'_{\mathsf{CPU}}, \mathsf{R/W}, L, \perp)$.
```

Figure 11: Description of the Step Circuit

[App17] Benny Applebaum. Garbled circuits as randomized encodings of functions: a primer. *IACR Cryptology ePrint Archive*, 2017:385, 2017.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

[BGL+15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 439–448. ACM Press, June 2015.

[BHR12a] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 134–153. Springer, Heidelberg, December 2012.

[BHR12b] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 784–796. ACM Press, October 2012.

[BL18] Fabrice Benhamouda and Huijia Lin. k-round multiparty computation from k-round oblivious transfer via garbled interactive circuits. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 500–532. Springer, Heidelberg, April / May 2018.

[BLSV18] Zvika Brakerski, Alex Lombardi, Gil Segev, and Vinod Vaikuntanathan. Anonymous ibe, leakage resilience and circular security from new assumptions. To appear in Eurocrypt, 2018. `https://eprint.iacr.org/2017/967`.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS*

93, pages 62–73. ACM Press, November 1993.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.

[CCHR16] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled RAM or: How to delegate your database. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 61–90. Springer, Heidelberg, October / November 2016.

[CDG$^+$17] Chongwon Cho, Nico Döttling, Sanjam Garg, Divya Gupta, Peihan Miao, and Antigoni Polychroniadou. Laconic receiver oblivious transfer and applications. *To appear in Crypto*, 2017.

[CH16] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In Madhu Sudan, editor, *ITCS 2016*, pages 169–178. ACM, January 2016.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 429–437. ACM Press, June 2015.

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

[DG17] Nico Döttling and Sanjam Garg. Identity based encryption from diffie-hellman assumptions. *To appear in Crypto*, 2017.

[DGHM18] Nico Dttling, Sanjam Garg, Mohammad Hajiabadi, and Daniel Masny. New constructions of identity-based and key-dependent message secure encryption schemes. To appear in PKC, 2018. `https://eprint.iacr.org/2017/978`.

[GGH$^+$13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[GGMP16] Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty RAM computation in constant rounds. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 491–520. Springer, Heidelberg, October / November 2016.

[GHL$^+$14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 405–422. Springer, Heidelberg, May 2014.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *55th FOCS*, pages 404–413. IEEE Computer Society Press, October 2014.

[GKK$^+$12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 513–524. ACM Press, October 2012.

[GKP$^+$13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 555–564. ACM Press, June 2013.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In Venkatesan Guruswami, editor, *56th FOCS*, pages 210–229. IEEE Computer Society Press, October 2015.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 449–458. ACM Press, June 2015.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

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

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.

[GS18a] Sanjam Garg and Akshayaram Srinivasan. Adaptively secure garbling with near optimal online complexity. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 535–565, 2018.

[GS18b] Sanjam Garg and Akshayaram Srinivasan. Two-round multiparty secure computation from minimal assumptions. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 468–499. Springer, Heidelberg, April / May 2018.

[HJO$^+$16] Brett Hemenway, Zahra Jafargholi, Rafail Ostrovsky, Alessandra Scafuro, and Daniel Wichs. Adaptively secure garbled circuits from one-way functions. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 149–178. Springer, Heidelberg, August 2016.

[HY16] Carmit Hazay and Avishay Yanai. Constant-round maliciously secure two-party computation in the RAM model. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 521–553. Springer, Heidelberg, October / November 2016.

[IKO$^+$11] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 406–425. Springer, Heidelberg, May 2011.

[JKK$^+$17] Zahra Jafargholi, Chethan Kamath, Karen Klein, Ilan Komargodski, Krzysztof Pietrzak, and Daniel Wichs. Be adaptive, avoid overcommitting. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 133–163, 2017.

[JW16] Zahra Jafargholi and Daniel Wichs. Adaptive security of Yao's garbled circuits. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 433–458. Springer, Heidelberg, October / November 2016.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In Rocco A.

Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 419–428. ACM Press, June 2015.

[KY18] Marcel Keller and Avishay Yanai. Efficient maliciously secure multiparty computation for ram. To appear in EUROCRYPT, 2018. `https://eprint.iacr.org/2017/981`.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 719–734. Springer, Heidelberg, May 2013.

[LO17] Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, pages 66–92, 2017.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[Mia16] Peihan Miao. Cut-and-choose for garbled RAM. Cryptology ePrint Archive, Report 2016/907, 2016. `http://eprint.iacr.org/2016/907`.

[ORS15] Rafail Ostrovsky, Silas Richelson, and Alessandra Scafuro. Round-optimal black-box two-party computation. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 339–358. Springer, Heidelberg, August 2015.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th ACM STOC*, pages 294–303. ACM Press, May 1997.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd ACM STOC*, pages 514–523. ACM Press, May 1990.

[WHC$^+$14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 191–202. ACM Press, November 2014.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.