Adaptively Secure Garbling Schemes for Parallel Computations

Kai-Min Chung[*] Luowen Qian[†]

Abstract

We construct the first adaptively secure garbling scheme based on standard public-key assumptions for garbling a circuit $C : \{0,1\}^n \mapsto \{0,1\}^m$ that simultaneously achieves a near-optimal online complexity $n + m + \mathsf{poly}(\lambda, \log |C|)$ (where λ is the security parameter) and *preserves the parallel efficiency* for evaluating the garbled circuit; namely, if the depth of C is d, then the garbled circuit can be evaluated in parallel time $d \cdot \mathsf{poly}(\log |C|, \lambda)$. In particular, our construction improves over the recent seminal work of [GS18], which constructs the first adaptively secure garbling scheme with a near-optimal online complexity under the same assumptions, but the garbled circuit can only be evaluated gate by gate in a sequential manner. Our construction combines their novel idea of linearization with several new ideas to achieve parallel efficiency without compromising online complexity.

We take one step further to construct the first adaptively secure garbling scheme for parallel RAM (PRAM) programs under standard assumptions that preserves the parallel efficiency. Previous such constructions we are aware of is from strong assumptions like indistinguishability obfuscation [ACC$^+$16]. Our construction is based on the work of [GOS18] for adaptively secure garbled RAM, but again introduces several new ideas to handle parallel RAM computation, which may be of independent interests. As an application, this yields the first constant round secure computation protocol for persistent PRAM programs in the malicious settings from standard assumptions.

[*]Institute of Information Science, Academia Sinica, Taipei 11529. Email: `kmchung@iis.sinica.edu.tw`. Partially supported by the Ministry of Science and Technology, Taiwan, under Grant no. MOST 106-2628-E-001-002-MY3 and the Academia Sinica Career Development Award under Grant no. 23-17.

[†]Department of Computer Science, Boston University, MA, USA. Email: `luowenq@bu.edu`.

Contents

1 Introduction

Garbled Circuits. The notion of garbled circuits were introduced by Yao [Yao82] for secure computations. Yao's construction of garbled circuits is secure in the sense that given a circuit C and an input x, the scheme gives out a garbled circuit $\widetilde{C}$ and a garbled input $\tilde{x}$ such that it only allows adversaries to recover $C(x)$ and nothing else. The notion of garbled circuits has found an enormous number of applications in cryptography. It is well established that garbling techniques is one of the important techniques in cryptography [BHR12, App17].

Garbled RAM. Lu and Ostrovsky [LO13] extended the garbling schemes to the RAM settings and its applications to delegating database and secure multiparty RAM program computation, and it has been an active area of research in garbling ever since [GHL$^+$14, GLOS15, GLO15]. Under this settings, it is possible to reduce the size of the garbled program to grow only linearly in the running time of the RAM program (and sometimes logarithmically in the size of the database), instead of the size of the corresponding circuit (which must grow linearly with the size of the database).

Parallel cryptography. It is a well established fact that parallelism is able to speed up computation, even exponentially for some problems. Yao's construction of garbled circuits is conceptually simple and inherently parallelizable. Being able to evaluate in parallel is more beneficial in the RAM settings where the persistent database can be very large, especially when it is applied to big data processing. The notion of parallel garbled RAM is introduced by Boyle et al [BCP16]. A black-box construction of parallel garbled RAM is known from one-way function [LO17].

Adaptively secure garbling. For certain applications of garbling, a stronger notion of *adaptive security* is usually required. We note that the notion of adaptive security is tightly related to efficiency.

For the circuit settings, the adversary is allowed to pick the input x to the program C after he has seen the garbled version of the program $\widetilde{C}$. In particular, for the circuit settings, we refer to the size of $\widetilde{C}$ as *offline complexity* and that of the garbled input $\tilde{x}$ as *online complexity*. The efficiency requirement says that the online complexity should not scale linearly with the size of the circuit[1]. Constructing adaptively secure garbling schemes for circuits with small online complexity has been an active area of investigation [HJO$^+$16, JW16, JKK$^+$17].

For the RAM settings, the adversary is allowed to adaptively pick multiple programs $\Pi_1, ..., \Pi_t$ and their respective inputs $x_1, ..., x_t$ to be executed on the same persistent database D, after he has seen the garbled version of the database $\widetilde{D}$, and having executed some garbled programs on the database and obtained their outputs $\Pi_i(x_i)$. Furthermore, he can choose his input after having seen the garbled program. The efficiency requirement is that the time for garbling the database, each program (and therefore the size of the garbled

[1]Note that without this efficiency requirement, any selectively secure garbled circuit can be trivially made adaptively secure, simply by sending everything only in the online phase. This also holds similarly for the RAM setting.

program) and the respective input should depend linearly only on the size of the database, the program, and the input respectively (up to poly logarithmic factors).

Parallel complexity of adaptively secure garbling. In two recent seminal works [GS18, GOS18], Garg et al. introduce an adaptively secure garbling scheme for circuits with near-optimal online complexity as well as for RAM programs. However, both constructions explicitly (using a linearization technique for circuits) or implicitly (serial execution of RAM programs) requires the evaluation process to proceed in a strict serial manner. Note that this would cause the parallel evaluation time of garbled circuits to blow up exponentially if the circuit depth is exponentially smaller than the size of the circuit. We also note that the linearization technique is their main technique for achieving near-optimal online complexity. On the other hand, such requirement seems to be at odds with evaluating the garbled version in parallel, which is something previous works [HJO+16] can easily achieve (however, Hemenway et al.'s construction has asymptotically greater online complexity). It's also not clear how to apply the techniques used in [GOS18] for adaptive garbled RAM to garble parallel RAM (PRAM) programs. In this work, we aim to find out whether such trade-off is inherent, namely,

Can we achieve adaptively secure garbling with parallel efficiency from standard assumptions?

1.1 Our Results

In this work, we obtained a construction of adaptively secure garbling schemes that allows for parallel evaluation, incurring only a logarithmic loss in the number of processors in online complexity based on the assumption that laconic oblivious transfer exists. Laconic oblivious transfer can be based on a variety of public-key assumptions [CDG+17, DG17, BLSV18, DGHM18]. More formally, our main results are:

Theorem 1.1. *Let λ be the security parameter. Assuming laconic oblivious transfer, there exists a construction of adaptively secure garbling schemes,*

- *for circuits C with optimal online communication complexity up to additive $\mathsf{poly}(\lambda, \log |C|)$ factors, and can be evaluated in parallel time $d \cdot \mathsf{poly}(\lambda, \log |C|)$ given w processors, where d and w are the depth and width of circuit C respectively;*
- *for PRAM programs on persistent database D, and can be evaluated in parallel time $T \cdot \mathsf{poly}(\lambda, \log M, \log |D|, \log T)$, where M is the number of processors and T is the parallel running time for the original program.*

This result closes the gap between parallel evaluation and online complexity for circuits, and also is the first adaptively secure garbling scheme for parallel RAM program from standard assumptions. Previous construction for adaptively secure garbled PRAM we are aware of is from strong assumptions like indistinguishability obfuscation [ACC+16].

We present our constructions formally in Section 4 for circuits and in Section 5 for PRAM.

1.2 Applications

In this section, we briefly mention some applications of our results.

Applications for parallelly efficient adaptive garbled circuits. Our construction of parallel adaptively secure garbled circuits can be applied the same way as already mentioned in previous works like [HJO$^+$16, GS18], e.g. to one-time program and compact functional encryption. Our result enables improved parallel efficiency for such applications.

Applications for adaptive garbled PRAM. This yields the first constant round secure computation protocol for persistent PRAM programs in the malicious settings from standard assumptions [GGMP16]. Prior works did not support persistence in the malicious setting. As a special case, this also allows for evaluating garbled PRAM programs on delegated persistent database.

2 Techniques

2.1 Parallelizing Garbled Circuits

Our starting point is to take Garg and Srinivasan's construction of adaptively secure garbled circuit with near-optimal online complexity [GS18] and allow it to be evaluated in parallel. Recall that the main idea behind their construction is to "linearize" the circuit before garbling it. Unfortunately, such transformation also ruins the parallel efficiency of their construction. We first explain why linearization is important to achieving near-optimal online complexity.

Pebbling game. Hemenway et al. [HJO$^+$16] introduced the notion of somewhere equivocal encryption, which enables us to equivocate a part of the garbled "gate" circuits and send them in the online phase. By using such technique, online complexity only needs to grow linearly in the maximum number of equivocated garbled gates at the same time over the entire hybrid argument, which could be much smaller than the length of the entire garbled circuit. Since an equivocated gate can be opened to be any gate, the simulator can simulate the gate according to the input chosen by the adversary, and send the simulated gate in the online phase. The security proof involves a hybrid argument, where in each step we change which gates we equivocate and show that this change is indistinguishable to the adversary. At a high level, this can be abstracted into a pebbling game.

Given a directed acyclic graph with a single sink, we can put or remove a pebble on a node if its every predecessor has a pebble on it or it has no predecessors. The game ends when there is a pebble on the unique sink. The goal of the pebble game is to minimize the maximum number of pebbles simultaneously on the graph throughout the game. In our case, the graph we need to pebble is what is called *simulation dependency graph*, where nodes represent garbled gates in the construction; and an edge from A to B represents that the input label for a piece B is hardcoded in A, thus to turn B into simulation mode, it is necessary to first turn A also into simulation mode. The simulation dependency graph directly corresponds to the circuit topology. The game terminates when the output gate is turned into simulation mode. As putting pebbles corresponds to equivocating the circuit in the online phase, the goal of the pebbling game also directly corresponds to the goal of minimizing online complexity.

Linearizing the circuit. It is known that there is a strong lower bound $\Omega(\frac{n}{\log n})$ for pebbling an arbitrary graph with n being the size of the graph [PTC76]. Since the circuits to be garbled can also be arbitrary, this means that the constructions of Hemenway et al. still have large online complexity for those "bad" circuits. Thus, Garg and Srinivasan pointed out that some change in the simulation dependency graph was required. In their work, they were able to change the simulation dependency graph to be a line, i.e. the simulation of any given garbled gate depends on only one other garbled gate. There's a good pebbling strategy using only $O(\log n)$ pebbles. On the other hand, using such technique also forces the evaluation to proceed sequentially, which would cause the parallel time complexity of wide circuits to blow up, in the worst case even exponentially.

We now describe how they achieved such linearization. In their work, instead of garbling the circuit directly, they "weakly" garble a special RAM program that evaluates the circuit. Specifically, this is done by having an external memory storing the values of all the intermediate wires and then transforming the circuit into a sequence of CPU step circuit, where each step circuit evaluates a gate and performs reads and writes to the memory to store the results. The step circuits are then garbled using Yao's garbling scheme and the memory is protected with one-time pad and laconic oblivious transfer (ℓOT). This garbling is weak since it does not protect the memory access pattern (which is fixed) and only concerns this specific type of program. Note that with this way, the input and output to the circuit can be revealed by revealing the one-time pad protecting the memory that store the circuit output, which only takes online complexity $n + m$.

Overview of our approach. A natural idea is that we can partially keep the linear topology, for which we know a good pebbling strategy; and at the same time, we would use M processors for each time step, each evaluating a gate in parallel. We then store the evaluation results by performing reads and writes on our external memory.

However, there are two challenges with this approach.

- **Parallel writes.** Read procedure in the original ℓOT scheme can be simply evaluated in parallel for parallel reads. On the other hand, since (as we will see later) the write procedure outputs an updated digest of the database, some coordination is obviously required, and simply evaluating writes in serial would result in a blow up in parallel time complexity. Therefore, we need to come up with a new parallel write procedure for this case.

- **Pebbling complexity.** Since now there are M gates being evaluated in parallel and looking ahead, they also need to communicate with each other to perform parallel writes, this will introduce complicated dependencies in the graph, and in the end, we could incur a loss in online complexity. Therefore, we must carefully layout our simulation dependency graph and find a good pebbling strategy for that graph.

2.1.1 Laconic OT

As mentioned earlier, we cannot use the write procedure in laconic oblivious transfer in a black-box way to achieve parallel efficiency. Thus, first we will elaborate on laconic oblivious transfer. Laconic oblivious transfer allows a receiver to commit to a large input via a short message of length λ. Subsequently, the sender responds with a single short message (which is also referred to as ℓOT ciphertext) to the receiver depending on dynamically chosen two messages m_0, m_1 and a location $L \in [|D|]$. The sender's response, enables the receiver to recover $m_{D[L]}$, while $m_{1-D[L]}$ remains computationally hidden. Note that the commitment does not hide the database and one commitment is sufficient to recover multiple bits from the database by repeating this process. ℓOT is frequently composed with Yao's garbled circuits to make a long process non-interactive. There, the messages will be chosen as input labels to the garbled circuit.

First, we briefly recall the original construction of ℓOT. The novel technique of laconic oblivious transfer was introduced in [CDG+17], where the scheme is constructed as a Merkle tree of "laconic oblivious transfer with factor-2 compression", which we denote as ℓOT_{const}, where the database is of length 2λ instead of being arbitrarily large. For the read procedure, we simply start at the root digest, traverse down the Merkle tree by using ℓOT_{const} to read out the digest for the next layer. Such procedure is then made non-interactive using Yao's garbled circuits (see Section 5.4 for details on this). For writes, similar techniques apply except that in the end, a final garbled circuit would take another set of labels for the digests to evaluate the updated root digest.

From the view of applying ℓOT to garbling RAM programs, an ℓOT scheme allows to compress a large database into a small digest of length λ that binds the entire database. In particular, given the digest, one can efficiently (in time only logarithmic in the size of the database) and repeatedly (ask the database holder to) read the database (open the commitment) or update the database and obtain the (correctly) updated digest. For both cases, as the evaluation results are returned as labels, the privacy requirement achieves "authentication", meaning the result has to be evaluated honestly as the adversary cannot obtain the other label.

Now, we will describe how we solve these two challenges.

2.1.2 Solving Parallel Writes

First attempt. Now we address how to parallelize ℓOT writes, in particular the garbled circuit evaluating the updated digest. First, we examine the task of designing a parallel algorithm with M processors that jointly compute the updated digest after writing M bits. At a high level, this can be done using the following procedure: all processors start from the bottom, make their corresponding modifications, and hash their ways up in the tree to compute the new digest; in each round, if two processors move to the same node, one of them is marked inactive and moved to the end using a sorting network. This intuitive parallel algorithm runs in parallel time $\mathsf{poly}(\log M, \log |C|, \lambda)$. By plugging such parallel algorithm back to the single write procedure for ℓOT, we obtain a parallel write procedure for ℓOT.

However, there are some issues for online complexity when we combine this intuitive algorithm with garbling and somewhere equivocal encryption. First, if we garble the entire parallel write circuit using Yao's garbling scheme, we would have to equivocate the entire parallel write circuit in the online phase at some point. Since the size of such circuit must be $\Omega(M)$, this leads to a large block length and we will get high online complexity. Therefore, we will have to split the parallel write circuit into smaller components and garble them separately so that we can equivocate only some parts of the entire write circuit in the online phase. However, this does not solve the problem completely, as in the construction of parallel writes for ℓOT given above, inter-CPU communications like sorting networks take place. In the end, this causes high pebbling complexity of $\Omega(M)$. This is problematic since M can be as large as the width of the circuit.

Block-writing ℓOT. To fix this issue, we note that for circuits, we can arbitrarily specify the memory locations for each intermediate wires, and this allows us to arrange the locations such that the communication patterns can be simplified to the extent that we can reduce the pebbling complexity to $O(\log M)$. One such good arrangement is moving all M updated locations into a single continuous block.

We give a procedure for handling such special case of updating the garbled database with ℓOT. Recall that in ℓOT, memory contents are hashed together using Merkle trees. Here, to simplify presentation, we assume the continuous block to be an entire subtree of the Merkle tree. In this case, it's easy to compute the digest of the entire subtree efficiently in parallel, after which we can just update the rest of the Merkle tree using a single standard but truncated writing procedure with time $\mathsf{poly}(\log |C|, \lambda)$, as we only need to pass and update the digest of the root of that sub-tree; and the security proof is analogous to that of a single write.

2.1.3 Pebbling Strategy

Before examining the pebbling strategy, we first give the description of the evaluation procedure and our transformed simulation dependency graph using the ideas mentioned in the previous section. In each round, M garbled circuits take the current database digest as input and each outputs a ℓOT ciphertext that allows the evaluator to obtain the input for a certain gate. Another garbled circuit would then take the input and evaluate the gate and output the label for the output for that gate. In order to hash together the output of M values for the gates we just evaluated, we use a Merkle tree of garbled circuits where each circuit would be evaluating a ℓOT hash with factor-2 compression. At the end of the Merkle tree, we would obtain the digest of the sub-tree we wish to update, which would then allow us to update the database and compute the updated digest. We can then use the updated digest to enable the evaluation of the next round.

Roughly, the pebbling graph we are dealing with is a line of "gadgets", and each gadget consists of a tree with children with an edge to their respective parents. One illustration of such gadget can be seen in Figure 9. One important observation here is that in order to start putting pebbles on any gadget, one only needs to put a pebble at the end of the previous gadget. Therefore, it's not hard to prove that the pebbling cost for the whole graph is the pebbling cost for a single gadget plus the pebbling cost for a line graph, whose length is the parallel time complexity of evaluating the circuit.

Pebbling line graph. Garg and Srinivasan used a pebbling strategy for pebbling line graphs with the number of pebbles logarithmic in the length of the line graph. Such strategy is optimal for the line graph. [Ben89]

Pebbling the gadget. For the gadget, the straightforward recursive strategy works very well:[2]

1. To put a pebble at the root, we first recursively put a pebble at its two children respectively;
2. Now we can put pebble at the root;
3. We again recursively remove the pebbles at its two children.

By induction, it's not hard to prove that such strategy uses the number of pebbles linear in the depth of the tree (note that at any given time, there can be at most 2 pebbles in each depth of the tree) and the number of steps is polynomial in the size of the graph.

Putting the two strategy together, we achieve online complexity $n+m+\mathsf{poly}(\lambda, \log|C|, \log M)$, where n, m is the length of the input and the output respectively. Note that M is certainly at most $|C|$, so the online complexity is in fact $n+m+\mathsf{poly}(\lambda, \log|C|)$, which matches the online complexity in [GS18].

2.2 Garbling Parallel RAM

Now we expand our previous construction of garbled circuits (which is a "weak" garbling of a special PRAM program) to garble more general PRAM programs, employing similar techniques from the seminal work of [GOS18]. We start by bootstrapping the garbling scheme into an adaptive garbled PRAM with unprotected memory access (UMA).

As with parallelizing adaptive garbled circuits, here we also face the issue of handling parallel writes. Note that here the previous approach of rearranging write locations would not work since due to the nature of RAM programs, the write locations can depend dynamically on the input. Therefore, we have to return to our first attempt of parallel writes and splitting the parallel evaluation into several circuits so that we can garble them separately for equivocation. Again, we run into the issue of communications leading to high pebbling complexity.

Solution: Parallel Checkpoints. Our idea is to instead put the parallel write procedure into the PRAM program and use a technique called "parallel checkpoints" to allow for arbitrary inter-CPU communications. At a high level, at the end of each parallel CPU step, we store all the CPUs' encrypted intermediary states into a second external memory and compute a digest using laconic oblivious transfer. Such digest can then act like a *checkpoint in parallel computation*, which is then used to retrieve the states back from the new database using another garbled circuit and ℓOT.

[2]This strategy is similar to the second strategy in [HJO$^+$16]. However, here the depth of the tree is only logarithmic in the number of processors so we can prevent incurring an exponential loss.

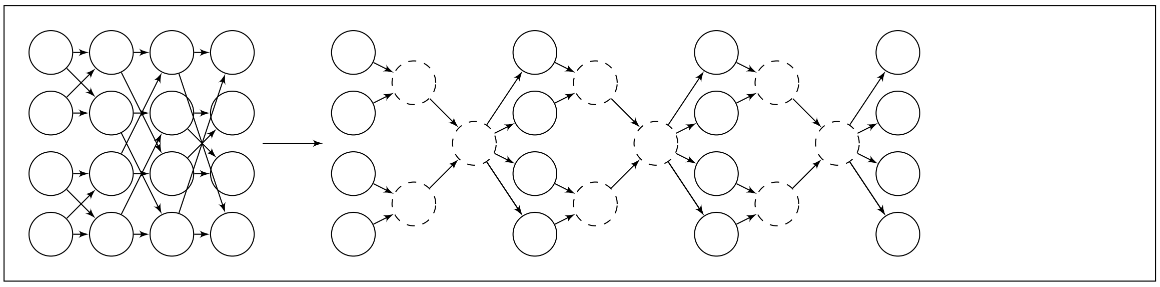

Figure 1: Transforming a toy sorting network using "parallel checkpoints." The undashed vertices corresponds to the step circuits that do the actual sorting.

To see how this change affects the simulation dependency graph and why it solves the complexity issue, consider the following toy example where we have a small sorting network, as seen in the left side of Figure 1. Note that applying the two pebbling strategeies from [HJO+16] directly on the untransformed network will result in an online complexity linear in either the number of processors M, or the running time T (and in this case also a security loss exponential in T). However, by doing the transformation as shown in Figure 1, we can pebble this graph with only $O(\log M)$ pebbles, by moving the pebble on the final node of each layer forward (and we can move the pebble forward by one layer using $O(\log M)$ pebbles). We can also see that using this change, the size of the garbled program will only grow by a factor of 2, and the parallel running time will only grow by a factor of $\log M$. In general, this transformation allows us to perform arbitrary inter-CPU communications without incuring large losses in online complexity, which resolves the issue.

For a more extended version of this construction, see Section 5.3.

Remark 2.1. *This technique would also work for parallelizing adaptive garbled circuits, but doing it this way would be way more complicated and less efficient.*

Pebbling game for parallel checkpoints. As mentioned above, such parallel checkpoints are implemented via creating a database using ℓOT. Thus the same strategy for pebbling the circuit pebble graph can be directly applied here. The key size of somewhere equivocal encryption is therefore only $\mathsf{poly}(\lambda, \log|D|, \log M, \log T)$.

With preprocessing and parallel checkpoints, we can proceed in a similar way to construct adaptively secure garbled PRAM with unprotected memory access. In order to bootstrap it from UMA to full security, the same techniques, i.e. timed encryption and oblivious RAM compiler from [GOS18] can be used in a similar way to handle additional complications in the RAM settings. In particular, we argue that the oblivious parallel RAM compiler from [BCP16] can be modified in the same way to achieve their strengthened notion of strong localized randomness in the parallel setting and handle the additional subtleties there. In the end, this allows us to construct a fully adaptively secure garbled PRAM.

3 Preliminaries

3.1 Garbled Circuits

In this section, we recall the notion of garbled circuits introduced by Yao [Yao82]. We will follow the same notions and terminologies as used in [CDG+17]. A circuit garbling scheme GC is a tuple of PPT algorithms $(\mathsf{GCircuit}, \mathsf{GCEval})$.

- $\tilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}, \{\mathsf{key}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}\right)$. It takes as input a security parameter λ, a circuit C, a set of labels $\mathsf{key}_{w,b}$ for all the input wires $w \in \mathsf{inp}(\mathsf{C})$ and $b \in \{0,1\}$. This procedure outputs a *garbled circuit* $\tilde{\mathsf{C}}$.
- $y \leftarrow \mathsf{GCEval}\left(\tilde{\mathsf{C}}, \{\mathsf{key}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}\right)$. Given a garbled circuit $\tilde{\mathsf{C}}$ and a garbled input represented as a sequence of input labels $\{\mathsf{key}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}$, GCEval outputs y.

Correctness. For correctness, we require that for any circuit C and input $x \in \{0,1\}^m$, where m is the input length to C, we have that

$$\Pr\left[\mathsf{C}(x) = \mathsf{GCEval}\left(\tilde{\mathsf{C}}, \{\mathsf{key}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}\right)\right] = 1,$$

where $\tilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}, \{\mathsf{key}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}\right)$.

Security. We require that there is a PPT simulator $\mathsf{GCircSim}$ such that for any C, x, and for $\{\mathsf{key}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}$ uniformly sampled,

$$\left(\tilde{\mathsf{C}}, \{\mathsf{key}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}\right) \overset{c}{\approx} \left(\mathsf{GCircSim}\left(1^\lambda, 1^{|\mathsf{C}|}, \{\mathsf{key}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}, y\right), \{\mathsf{key}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}\right),$$

where $\tilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}, \{\mathsf{key}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}\right)$ and $y = \mathsf{C}(x)$.

Parallel efficiency. For parallel efficiency, we require that the parallel runtime of $\mathsf{GCircuit}$ on a PRAM machine with M processors is $\mathsf{poly}(\lambda) \cdot |C|/M$ if $|C| \geq M$, and the parallel runtime of GCEval on a PRAM machine with w processors is $\mathsf{poly}(\lambda) \cdot d$, where w, d is the width and depth of the circuit respectively.

3.2 Somewhere Equivocal Encryption

In this section, we recall the definition of Somewhere Equivocal Encryption from the work of [HJO+16].

Definition 3.1. *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 PPT algorithms* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{SimEnc}, \mathsf{SimDec})$ *such that:*

- $\mathsf{key} \leftarrow \mathsf{KeyGen}(1^\lambda)$*: It takes as input the security parameter λ and outputs a key* key.
- $\bar{c} \leftarrow \mathsf{Enc}(\mathsf{key}, \bar{m})$*: It takes as input a key* key *and a vector of messages $\bar{m} = m_1...m_n$ with each $m_i \in \{0,1\}^s$ and outputs a ciphertext $\bar{c}$.*

```
$\mathsf{SimEncExpt}^{b\in\{0,1\}}(1^\lambda, \mathcal{A})$
    Let $I_0 = I$ and $I_1 = I \cup \{j\}$
    $(\mathsf{st}, \bar{c}) \leftarrow \mathsf{SimEnc}((m_i)_{i\notin I_b}, I_b)$
    $((m_i)_{i\in I}, \mathsf{st}') \leftarrow \mathcal{A}_1(\bar{c})$
    $\mathsf{key} \leftarrow \mathsf{SimKey}(\mathsf{st}, (m_i)_{i\in I_b})$
    Output $\mathcal{A}_2(\mathsf{st}', \mathsf{key})$
```

Figure 2: Simulated Encryption Experiment

- $\bar{m} \leftarrow \mathsf{Dec}(\mathsf{key}, \bar{c})$*: It is a deterministic algorithm that takes as input a key* key *and a ciphertext* $\bar{c}$ *and outputs a vector of messages* $\bar{m} = m_1 ... m_n$*.*
- $(\mathsf{st}, \bar{c}) \leftarrow \mathsf{SimEnc}((m_i)_{i\notin I}, I)$*: It takes as input a set of indices* $I \subseteq [n]$ *and a vector of messages* $(m_i)_{i\notin I}$ *and outputs a ciphertext* $\bar{c}$ *and a state* st*.*
- $\mathsf{key}' \leftarrow \mathsf{SimKey}(\mathsf{st}, (m_i)_{i\in I})$*: It takes as input the state information* st *and a vector of messages* $(m_i)_{i\in I}$ *and outputs a key* key'*.*

It is required to satisfy the following properties:

Correctness. *For every* $\mathsf{key} \leftarrow \mathsf{KeyGen}(1^\lambda)$*, every* $\bar{m} \in \{0,1\}^{s\times n}$*, we require that*

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

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

Security. *For any non-uniform PPT adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*, for any* $I \subseteq [n]$ *s.t.* $|I| \leq t$*,* $j \in [n] - I$ *and vector* $(m_i)_{i\notin I}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *s.t.*

$$|\Pr[\mathsf{SimEncExpt}^0(1^\lambda, \mathcal{A}) = 1] - \Pr[\mathsf{SimEncExpt}^1(1^\lambda, \mathcal{A}) = 1]| \leq \mathsf{negl}(\lambda),$$

where $\mathsf{SimEncExpt}^0$ *and* $\mathsf{SimEncExpt}^1$ *are described in Figure 2.*

Theorem 3.2 ([HJO+16])**.** *Assuming the existence of one-way functions, there exists a somewhere equivocal encryption scheme for any polynomial message-length* n*, block-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 Parallel RAM Programs

We follow the formalization of parallel RAM (PRAM) programs used in [LO17]. A M parallel random-access machine is a collection of M processors $\mathsf{CPU}_1, ..., \mathsf{CPU}_m$, having concurrent access to a shared external memory D.

A PRAM *program* Π, given input $x_1, ..., x_M$, provides instructions to the CPUs that can access to the shared memory D. The CPUs execute the program until a halt state is reached, upon which all CPUs collectively output $y_1, ..., y_M$.[3]

Here, we formalize each processor as a step circuit, i.e. for each step, CPU_i evaluates the circuit $C^{\Pi}_{\mathsf{CPU}_i}(\mathsf{state}, \mathsf{wData}) = (\mathsf{state}', \mathsf{R/W}, L, \mathsf{rData})$. This circuit takes as input the current CPU state state and the data rData read from the database, and it outputs an updated state state', a read or write bit $\mathsf{R/W}$, the next locations to read/write L, and the data wData to write to that location. We allow each CPU to request up to γ bits at a time, therefore here $\mathsf{rData}, \mathsf{wData}$ are both bit strings of length γ. For our purpose, we assume $\gamma \geq 2\lambda$. The (parallel) time complexity T of a PRAM program Π is the number of time steps taken to evaluate Π before the halt state is reached.

We note that the notion of parallel random-access machine is a commonly used extension of Turing machine when one needs to examine the concrete parallel time complexity of a certain algorithm.

Memory access patterns. The memory access pattern of PRAM program $\Pi(x)$ is a sequence $(\mathsf{R/W}_i, L_i)_{i\in[T]}$, each element represents at time step i, a read/write $\mathsf{R/W}_i$ was performed on memory location L_i.

3.4 Sorting Networks

Our construction of parallel ℓOT uses *sorting networks*, which is a fixed topology of comparisons for sorting values on n wires. In our instantiation, n equals the number of processors M in the PRAM model. As PRAM can simulate circuits efficiently, on a high level, a sorting network of depth d corresponds to a parallel sorting algorithm with parallel time complexity $O(d)$. As mentioned previously, the topology of the sorting network is not relevant to our construction.

Theorem 3.3 ([AKS83]). *There exists an n-wire sorting network of depth $O(\log n)$.*

3.5 Laconic Oblivious Transfer

Definition 3.4 ([CDG$^+$17]). *An updatable laconic oblivious transfer (ℓOT) scheme consists of four algorithms* crsGen, Hash, Send, $\mathsf{Receive}$, $\mathsf{SendWrite}$, $\mathsf{ReceiveWrite}$.

- $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$. *It takes as input the security parameter 1^λ and outputs a common reference string* crs.
- $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$. *It takes as input a common reference string* crs *and a database $D \in \{0,1\}^*$ and outputs a digest* digest *of the database and a state* $\hat{\mathsf{D}}$.

[3]Similarly, here we assume the program is deterministic. We can allow for randomized execution by providing it random coins as input.

- $\mathsf{e} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$. *It takes as input a common reference string* crs*, a digest* digest*, a database location $L \in \mathbf{N}$ and two messages m_0 and m_1 of length λ, and outputs a ciphertext* e.

- $m \leftarrow \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L)$. *This is a RAM algorithm with random read access to* $\hat{\mathsf{D}}$. *It takes as input a common reference string* crs*, a ciphertext* e*, and a database location $L \in \mathbf{N}$. It outputs a message m.*

- $\mathsf{e_w} \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{digest}, \{L_k\}_{k\in[M]}, \{b_k\}_{k\in[M]}, \{m_{j,c}\}_{j\in[\lambda],c\in\{0,1\}}\right)$. *It takes as input the common reference string* crs*, a digest* digest*, M locations $\{L_k\}_k$ with the corresponding bits $\{b_k\}_k$, and λ pairs of messages $\{m_{j,c}\}_{j\in[\lambda],c\in\{0,1\}}$, where each $m_{j,c}$ is of length λ. It outputs a ciphertext* $\mathsf{e_w}$.

- $\{m_j\}_{j\in[\lambda]} \leftarrow \mathsf{ReceiveWrite}^{\tilde{\mathsf{D}}}(\mathsf{crs}, \{L_k\}_{k\in[M]}, \{b_k\}_{k\in[M]}, \mathsf{e_w})$. *This is a RAM algorithm with random read/write access to* $\tilde{\mathsf{D}}$. *It takes as input the common reference string* crs*, M locations $\{L_k\}_{k\in[M]}$ and bits to be written $\{b_k\}_{k\in[M]}$ and a ciphertext* $\mathsf{e_w}$. *It updates the state* $\tilde{\mathsf{D}}$ *(such that $D[L_k] = b_k$ for every $k \in [M]$) and outputs messages $\{m_j\}_{j\in[\lambda]}$.*

It is required to satisfy the following properties:

- ***Correctness:*** *For any database D of size at most* $\mathsf{poly}(\lambda)$ *for any polynomial function* $\mathsf{poly}(\cdot)$*, any memory location $L \in [|D|]$, and any pair of messages $(m_0, m_1) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$ that*

$$\Pr\left[m = m_{D[L]} \;\middle|\; \begin{array}{r} \mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda) \\ (\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D) \\ \mathsf{e} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1) \\ m \leftarrow \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L) \end{array} \right] = 1,$$

where the probability is taken over the random choices made by crsGen *and* Send.

- ***Correctness of Writes:*** *For any database D of size at most* $\mathsf{poly}(\lambda)$ *for any polynomial function* $\mathsf{poly}(\cdot)$*, any M memory locations $\{L_j\}_j \in [|D|]^M$ and any bits $\{b_j\}_j$, and any pairs of messages $\{m_{j,c}\}_{j,c} \in \{0,1\}^{2\lambda^2}$, let D^* be the database to be D after making the modifications $D[L_j] \leftarrow b_j$ for $j = 1, ..., M$, we require that*

$$\Pr\left[\begin{array}{l} m'_j = m_{j,D[L]} \\ \forall j \in [\lambda] \end{array} \;\middle|\; \begin{array}{r} \mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda) \\ (\mathsf{d}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D) \\ (\mathsf{d}^*, \hat{\mathsf{D}^*}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*) \\ \mathsf{e} \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{d}, \{L_k\}_k, \{b_k\}_k, \{m_{j,c}\}_{j,c}\right) \\ \{m'_j\}_j \leftarrow \mathsf{ReceiveWrite}^{\tilde{\mathsf{D}}}(\mathsf{crs}, \{L_k\}_k, \{b_k\}_k, \mathsf{e}) \end{array} \right] = 1,$$

where the probability is taken over the random choices made by crsGen *and* Send.

```
$\mathsf{SenPrivExpt}^{\tau\in\{\text{real, ideal}\}}(1^\lambda, \mathcal{A})$
    $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$
    $(D, L, m_0, m_1, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{crs})$
    $(d, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
    If $\tau$ is $\mathsf{real}$, $\mathsf{e} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{d}, L, m_0, m_1)$
    If $\tau$ is $\mathsf{ideal}$, $\mathsf{e} \leftarrow \ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]})$
    Output $\mathcal{A}_2(\mathsf{st}, \mathsf{e})$
```

Figure 3: Sender Privacy Security Game

```
$\mathsf{SenPrivWriteExpt}^{\tau\in\{\text{real, ideal}\}}(1^\lambda, \mathcal{A})$
    $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$
    $(D, M, \{L_j\}_{j\in[M]}, \{m_{j,c}\}_{j,c}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{crs})$
    $(d, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
    $(d^*, \widehat{D}^*) \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*)$ where $D^*$ is $D$ after making the modifications $D[L_j] \leftarrow b_j$
for $j = 1, ..., M$
    If $\tau$ is $\mathsf{real}$, $\mathsf{e} \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{d}, \{L_k\}_k, \{b_k\}_k, \{m_{j,c}\}_{j,c}\right)$
    If $\tau$ is $\mathsf{ideal}$, $\mathsf{e} \leftarrow \ell\mathsf{OTSimWrite}(\mathsf{crs}, D, \{L_k\}_k, \{m_{j,d_j^*}\}_j)$
    Output $\mathcal{A}_2(\mathsf{st}, \mathsf{e})$
```

Figure 4: Sender Privacy Security Game for Writes

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

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

  where $\mathsf{SenPrivExpt}^{\mathsf{real}}$ *and* $\mathsf{SenPrivExpt}^{\mathsf{ideal}}$ *are described in Figure 3.*

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

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

  where $\mathsf{SenPrivWriteExpt}^{\mathsf{real}}$ *and* $\mathsf{SenPrivWriteExpt}^{\mathsf{ideal}}$ *are described in Figure 4.*

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

It is also helpful to introduce the ℓOT scheme with factor-2 compression, which is used in ℓOT's original construction [CDG+17].

Definition 3.5. *An ℓOT scheme with factor-2 compression ℓOT_{const} is an ℓOT scheme where the database D has to be of size 2λ.*

Remark 3.6. *The sender privacy requirement here is from [GS18]. It requires* crs *to be given to the adversary before the adversary chooses his challenge instead of after, and is therefore stronger than the original security requirement [CDG+17]. But we note that in the security proof of laconic oblivious transfer, such adaptive security requirement can be directly reduced to adaptive security for ℓOT_{const}. And in the construction of [CDG+17], in every hybrid,* crs *is generated either truthfully, or generated statistically binding to one of 2λ possible positions. Therefore, we will incur at most a $1/2\lambda$ loss in the security reduction, simply by guessing which position we need to bind to in those hybrids. This also applies to the sender privacy for parallel writes we will discuss later.*

Theorem 3.7 ([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 laconic oblivious transfer.*

4 Adaptive Garbled Circuits Preserving Parallel Runtime

In this section, we construct an adaptively secure garbling scheme for circuits that allows for parallel evaluation without compromising near-optimal online complexity. We follow the definition of adaptive garbled circuits from [HJO+16].

Definition 4.1. *An adaptive garbling scheme for circuits is a tuple of PPT algorithms* $(\mathsf{AdaGCircuit}, \mathsf{AdaGInput}, \mathsf{AdaEval})$ *such that:*

- $(\tilde{C}, \mathsf{st}) \leftarrow \mathsf{AdaGCircuit}\left(1^\lambda, C\right)$. *It takes as input a security parameter λ, a circuit $C : \{0,1\}^n \mapsto \{0,1\}^m$ and outputs a garbled circuit $\tilde{C}$ and state information* st.
- $\tilde{x} \leftarrow \mathsf{AdaGInput}(\mathsf{st}, x)$*: It takes as input the state information* st *and an input $x \in \{0,1\}^n$ and outputs the garbled input $\tilde{x}$.*
- $y \leftarrow \mathsf{AdaEval}(\tilde{C}, \tilde{x})$. *Given a garbled circuit $\tilde{\mathsf{C}}$ and a garbled input $\tilde{x}$,* $\mathsf{AdaEval}$ *outputs $y \in \{0,1\}^m$.*

Correctness. *For any $\lambda \in \mathbf{N}$ circuit $\mathsf{C} : \{0,1\}^n \mapsto \{0,1\}^m$ and input $x \in \{0,1\}^n$, we have that*

$$\Pr\left[C(x) = \mathsf{AdaEval}(\tilde{C}, \tilde{x})\right] = 1,$$

where $(\tilde{C}, \mathsf{st}) \leftarrow \mathsf{AdaGCircuit}\left(1^\lambda, C\right)$ *and* $\tilde{x} \leftarrow \mathsf{AdaGInput}(\mathsf{st}, x)$.

Adaptive Security. *There is a PPT simulator* $\mathsf{AdaGSim} = (\mathsf{AdaGSimC}, \mathsf{AdaGSimIn})$ *such that, for any non-uniform PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$ there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that*

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

where $\mathsf{AdaGCExpt}^{\mathsf{real}}$ *and* $\mathsf{AdaGCExpt}^{\mathsf{ideal}}$ *are described in Figure 5.*

```
$\mathsf{AdaGCExpt}^{\tau \in \{\mathsf{real},\ \mathsf{ideal}\}}[1^\lambda, \mathcal{A}]$
    $(C, s) \leftarrow \mathcal{A}_1(1^\lambda)$
    If $\tau$ is $\mathsf{real}$, $(\tilde{C}, \mathsf{st}) \leftarrow \mathsf{AdaGCircuit}\left(1^\lambda, C\right)$
    If $\tau$ is $\mathsf{ideal}$, $(\tilde{C}, \mathsf{st}) \leftarrow \mathsf{AdaGSimC}\left(1^\lambda, 1^{|C|}\right)$
    $(x, s) \leftarrow \mathcal{A}_2(s, \tilde{C})$
    If $\tau$ is $\mathsf{real}$, $\tilde{x} \leftarrow \mathsf{AdaGInput}(\mathsf{st}, x)$
    If $\tau$ is $\mathsf{ideal}$, $\tilde{x} \leftarrow \mathsf{AdaGSimIn}(\mathsf{st}, C(x))$
    Output $\mathcal{A}_3(s, \tilde{x})$
```

Figure 5: Adaptive Security Game of Adaptive Garble Circuits

Online Complexity. *The running time of* $\mathsf{AdaGInput}$ *is called the online computational complexity and* $|\tilde{x}|$ *is called the online communication complexity. We require that the online computational complexity does not scale linearly with the size of the circuit* $|C|$.

Furthermore, we call the garbling scheme is parallelly efficient, if the algorithms are given as probabilistic PRAM programs with M processors, and the parallel runtime of $\mathsf{AdaGCircuit}$ is $\mathsf{poly}(\lambda) \cdot |C|/M$ if $|C| \geq M$, the parallel runtime of $\mathsf{AdaGInput}$ on a PRAM machine to be $n/M \cdot \mathsf{poly}(\lambda, \log |C|)$, and the parallel runtime of $\mathsf{AdaEval}$ is $\mathsf{poly}(\lambda) \cdot d$ if $M \geq w$, where w, d is the width and depth of the circuit respectively.

4.1 Construction Overview

First, we recall the construction of [GS18], which we will use as a starting point. At a high level, their construction can be viewed as a "weak" garbling of a special RAM program that evaluates the circuit.

In the ungarbled world, a database D is used as RAM to store all the wires (including input, output, and intermediate wires). Initially, D only holds the input and everything else is uninitialized. In each iteration, the processor takes a gate, read two bits according to the gate, evaluate the gate, and write the output bit back into the database. Finally, after all iterations are finished, the output of the circuit is read from the database.

In the garbled world, the database D will be hashed as $\hat{D}$ using ℓOT and protected with an one-time pad r as ℓOT does not protect its memory content. The evaluation process is carried out by a sequence of Yao's garbled circuits and laconic OT "talking" to each other. In each iteration, two read operations correspond to a selectively secure garbled circuit, which on given digest as input, outputs two ℓOT read ciphertexts that the evaluator can decrypt to the input label for the garbled gate, which is a selectively secure garbled circuit that unmasks the input, evaluates the gate, and then unmasks the output. To store the output, the garbled gate generates a ℓOT block-write ciphertext, which also enables the evaluator to obtain the input labels for the updated digest in the next iteration. This garbled RAM program is then encrypted using a somewhere equivocal encryption, after which it is given to the adversary as the garbled circuit. On given input x, we generate the protected database $\hat{D}$ and compute the input labels for the initial digest, and we give out the labels, the masked input, the decryption key, and masks for output in the database.

This garbling is weak as in it only concerns a particular RAM program, and it does not protect the memory access pattern, but it is sufficient for the adaptive security requirement of garbled circuits as the pattern is fixed and public. As we will see in the security proof that online complexity is tightly related to the pebbling complexity of a pebbling game. The pebbling game is played on a simulation dependency graph, where pieces of garbled circuits in the construction correspond to nodes and hardwiring of input labels correspond to edges. As the input labels for every selectively secure circuit is only hardcoded in the previous circuit, the simulation dependency is a line and there is a known good pebbling strategy.

To parallelize this construction, we naturally wish to evaluate M gates in parallel using a PRAM program instead of evaluating sequentially. This way, we preserve its mostly linear structure, for which we know a good pebbling strategy. Reading from the database is inherently parallelizable, but writing is more problematic as the processors need to communicate with each other to compute the updated digest and we need to be more careful.

4.2 Block-writing Laconic OT

Recall from Section 2.1 that we cannot hope to use ℓOT as a black box in parallel, thus we first briefly recall the techniques used in [CDG+17] to bootstrap an ℓOT scheme with factor-2 compression ℓOT_{const} into a general ℓOT scheme with an arbitrary compression factor.

Consider a database D with size $|D| = 2^d \cdot \lambda$. In order to obtain a hash function with arbitrary (polynomial) compression factor, it's natural to use a Merkle tree to compress the database. The Hash function outputs $(\mathsf{digest}, \hat{\mathsf{D}})$, where $\hat{\mathsf{D}}$ is the Merkle tree and digest is the root of the tree. Using ℓOT_{const} combined with a Merkle tree, the sender is able to traverse down the Merkle tree, simply by using $\ell OT_{\mathsf{const}}.\mathsf{Send}$ to obtain the digest for any child he wishes to, until he reaches the block he would like to query. For writes, the sender can read out all the relevant neighbouring digests from the Merkle tree and compute the updated digest using the information. In order to compress the round complexity down to 1 from d, we can use Yao's garble circuit to garble $\ell OT_{\mathsf{const}}.\mathsf{Send}$ so that the receiver can evaluate it for the sender, until he gets the final output. On a high level, the receiver makes the garbled circuits and ℓOT_{const} talk to each other to evaluate the read/write ciphertexts.

As mentioned in Section 2.1, we wish to construct a block-write procedure such that the following holds:

- The parallel running time should be $\mathsf{poly}(\lambda, \log |C|)$;
- For near-optimal online complexity, both the size of each piece of the garbled circuit and the pebbling complexity needs to be $\mathsf{poly}(\lambda, \log |C|)$.

Note that changing the ciphertext to contain all M bits directly do not work in this context, as now the write ciphertext would be of length $\Omega(M)$, therefore the garbled circuit generating it must be of length $\Omega(M)$, which violates what we wish to have. The way to fix this is to instead let the ciphertext only hold the digest of the sub-tree, and the block write ciphertext simply needs to perform a "partial" write to obtain the updated digest, therefore its size is no larger than an ordinary write ciphertext. As it turns out, a tree-like structure

```
$\ell\mathsf{OTSimWriteBlock}(\mathsf{crs}, D, L, \{b_j\}_{j\in[\lambda]}, \{m_{j,\mathsf{digest}^*_j}\}_{j\in[\lambda]})$
    Output $\ell\mathsf{OTSimWrite}(\mathsf{crs}, D, \{L||j\}_{j\in[\lambda]}, \{b_j\}_{j\in[\lambda]}, \{m_{j,\mathsf{digest}^*_j}\}_{j\in[\lambda]})$
```

Figure 6: Block-writing security simulator

in the simulation dependency graph also has good pebbling complexity and we can obtain the sub-tree digest using what we call a garbled Merkle tree, which we will construct in the next section. This way, we resolve all the issues.

Now, we first direct our attention back to constructing block-writes. Formally, we will construct two additional algorithms for updatable laconic oblivious transfer that handles a special case of parallel writes. As we will see later, these algorithms can be used to simplify the construction of adaptive garbled circuit.

- $\mathsf{e_w} \leftarrow \mathsf{SendWriteBlock}\left(\mathsf{crs}, \mathsf{digest}, L, \mathsf{d}, \{m_{j,c}\}_{j\in[\lambda],c\in\{0,1\}}\right)$. It takes as input the common reference string crs, a digest digest, a location prefix $L \in \{0,1\}^P$ with length $P \leq \log|D|$ and the digest of the subtree d to be written to location $L00...0, L00...1, ..., L11...1$, and λ pairs of messages $\{m_{j,c}\}_{j\in[\lambda],c\in\{0,1\}}$, where each $m_{j,c}$ is of length λ. It outputs a ciphertext $\mathsf{e_w}$.

- $\{m_j\}_{j\in[\lambda]} \leftarrow \mathsf{ReceiveWriteBlock}^{\tilde{\mathsf{D}}}(\mathsf{crs}, L, \{b_k\}_{k\in[2^M]}, \mathsf{e_w})$. This is a RAM algorithm with random read/write access to $\tilde{\mathsf{D}}$. It takes as input the common reference string crs, M locations $\{L_k\}_{k\in[M]}$ and bits to be written $\{b_k\}_{k\in[M]}$ and a ciphertext $\mathsf{e_w}$. It updates the state $\tilde{\mathsf{D}}$ (such that $D[L_k] = b_k$ for every $k \in [M]$) and outputs messages $\{m_j\}_{j\in[\lambda]}$.

The formal construction of block-writing ℓOT is as follows:

```
• $\mathsf{SendWriteBlock}\left(\mathsf{crs}, \mathsf{digest}, L, \mathsf{d}, \{m_{j,c}\}_{j\in[\lambda],c\in\{0,1\}}\right)$
    Reinterpret the $\ell OT$ Merkle tree by truncating at the $|L|$-th layer
    Output $\ell OT.\mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{digest}, L, \mathsf{d}, \{m_{j,c}\}_{j\in[\lambda],c\in\{0,1\}}\right)$

• $\mathsf{ReceiveWriteBlock}^{\tilde{\mathsf{D}}}(\mathsf{crs}, L, \{b_k\}_{k\in[2^M]}, \mathsf{e_w})$
    Compute the digest $\mathsf{d}$ of database $\{b_k\}_{k\in[2^M]}$
    Reinterpret the $\ell OT$ Merkle tree by truncating at the $|L|$-th layer and $\tilde{\mathsf{D}}$ as the corresponding truncated version of the database
    $\mathsf{Label} \leftarrow \ell OT.\mathsf{ReceiveWrite}^{\tilde{\mathsf{D}}}(\mathsf{crs}, L, \{b_k\}_{k\in[2^M]}, \mathsf{e_w})$
    Update $\tilde{\mathsf{D}}$ at block location $L$ using data $\{b_k\}_{k\in[2^M]}$
    Output $\mathsf{Label}$
```

We require similar security and efficiency requirements for block-writing ℓOT. It's not hard to see that the update part of ReceiveWriteBlock can be evaluated efficiently in parallel (and the call to normal ReceiveWrite only needs to run once), and the security proof can be easily reduced to that of SendWrite.

```
Hashing Sub-Circuit $\mathsf{C}$
  Hardwired Values/Circuit: $H, \mathsf{Keys}$
  Input: $x \in \{0,1\}^{2\lambda}$
  Output $\mathsf{Keys}_{H(x)}$
```

Figure 7: Hashing Sub-Circuit

4.3 Garbled Merkle Tree

We will now describe an algorithm called garbled Merkle tree. Roughly speaking, a garbled Merkle tree is a binary tree of garbled circuits, where each of the circuit takes arbitrary 2λ bits as input and outputs the labels of λ bit digest. Looking ahead, this construction allows for exponentially smaller online complexity compared to simply garbling the entire hash circuit when combined with adaptive garbling schemes we will construct later, since its tree structure allows for small pebbling complexity.

A garbled Merkle tree has very similar syntax as the one for garbled circuit. It consists of 2 following PPT algorithms:

- $\mathsf{GHash}(1^\lambda, H, \{\mathsf{Key}_i\}_{i\in[|D|]}, \{\mathsf{Key}'_i\}_{i\in[\lambda]})$: it takes as input a security parameter λ, a hashing circuit H that takes 2λ bits as input and outputs λ bits, keys $\{\mathsf{Key}_i\}_{i\in[|D|]}$ for all bits in the database D and $\{\mathsf{Key}'_i\}_{i\in[\lambda]}$ for all output bits

```
  $\mathsf{Keys}_1 \leftarrow \{\mathsf{Key}'_i\}_{i\in[\lambda]}$
  Sample $\{\mathsf{Keys}_i\}_{i=2,\dots,|D|/\lambda-1}$
  $\{\mathsf{Keys}_i\}_{i=|D|/\lambda,\dots,2|D|/\lambda-1} \leftarrow \{\mathsf{Key}_i\}_{i\in[|D|]}$
  For $i = 1$ to $|D|/\lambda - 1$ do
    $\tilde{C}_i \leftarrow \mathsf{GCircuit}(1^\lambda, \mathsf{C}[H, \mathsf{Keys}_i], (\mathsf{Keys}_{2i}, \mathsf{Keys}_{2i+1}))$
  Output $\{\tilde{C}_i\}_{i\in[|D|/\lambda-1]}$
```

  The circuit C here is given in Figure 7.

- $\mathsf{GHEval}\left(\{\tilde{C}_i\}, \{\mathsf{lab}_i\}_{i\in[|D|]}\right)$: it takes as input the garbled circuits $\{\tilde{C}_i\}_{i\in[|D|/\lambda-1]}$ and input labels for the database $\{\mathsf{lab}_i\}_{i\in[|D|]}$

```
  $\{\mathsf{Label}_i\}_{i=|D|/\lambda,\dots,2|D|/\lambda-1} \leftarrow \{\mathsf{lab}_i\}_{i\in[|D|]}$
  For $i = |D|/\lambda - 1$ down to 1 do
    $\mathsf{Label}_i \leftarrow \mathsf{GCEval}(\tilde{C}_i, (\mathsf{Label}_{2i}, \mathsf{Label}_{2i+1}))$
  Output $\mathsf{Label}_1$
```

Later, we will also invoke this algorithm in garbled PRAM for creating parallel checkpoints.

4.4 Construction

We will now give the construction of our adaptive garbled circuits. Let ℓOT be a laconic oblivious transfer scheme, $(\mathsf{GCircuit}, \mathsf{GCEval})$ be a garbling scheme for circuits, $(\mathsf{GHash}, \mathsf{GHEval})$ be a garbling scheme for Merkle trees, and SEE be a somewhere equivocal encryption scheme with block length $\mathsf{poly}(\lambda, \log |C|)$ to be the maximum size of garbled circuits $\left\{\widetilde{\mathsf{C}}^{\mathsf{eval}}_{i,k}, \widetilde{\mathsf{C}}^{\mathsf{hash}}_{i,j}, \widetilde{\mathsf{C}}^{\mathsf{write}}_{i}\right\}$, message length $2M\ell = O(|C|^2)$ (we will explain ℓ shortly after) and equivocation parameter $\log \ell + 2\log M + O(1)$ (the choice comes from the security proof).

Furthermore, we assume both M and λ is a power of 2 and λ divides M. We also have a procedure $\{P_i\}_{i\in[\ell]} \leftarrow \mathsf{Partition}(C, M)$ (as an oracle) that partition the circuit's wires $1, 2, ..., |C|$ into ℓ continuous partitions of size M, such that for any partition P_i, its size is at most M (allowing a few extra auxilary wires and renumbering wires), and every gate in the partition can be evaluated in parallel once every partition P_j with $j < i$ has been evaluated. Clearly $d \le \ell \le |C|$, but it's also acceptable to have a sub-optimal partition to best utilize the computational resources on a PRAM machine. We assume the input wires are put in partition 0. This preprocessing is essentially scheduling the evaluation of the circuit to a PRAM machine and it is essential to making our construction's online complexity small.

We now give an overview of our construction. At a high level, instead of garbling the circuit directly, our construction can be viewed as a garbling of a special PRAM program that evaluates the circuit in parallel. The database D will be hashed as $\hat{D}$ using ℓOT and protected with an one-time pad r as ℓOT does not protect its memory content. In each iteration, two read operations for every processor correspond to two selectively secure garbled circuits, which on given digest as input, outputs a ℓOT read ciphertext that generates the input label for the garbled gate; the garbled gate unmasks the input, evaluates the gate, and then output the masked output of the gate. After all M processors have done evaluating their corresponding gates, a garbled Merkle tree will take their outputs as input to obtain the digest for the M bits of output, and then generate a ℓOT block-write ciphertext to store the outputs into the database. During evaluation, this block-write ciphertext can be used to obtain the input labels for the read circuits in the next iteration. This garbled PRAM program is then encrypted using a somewhere equivocal encryption, after which it is given to the adversary as the garbled circuit. On given input x, we generate the protected database $\hat{D}$ and compute the input labels for the initial digest, and we give out the labels, the masked inputs, the decryption key, and masks for outputs in the database.

Now we formally present the construction. Inside the construction, we omit $k \in [M]$ when the context is clear. It might also be helpful to see Figure 9 for how the garbled circuits are organized.

```
• $\mathsf{AdaGCircuit}^{\mathsf{Partition}}(1^\lambda, C)$:
    $\mathsf{crs} \leftarrow \ell OT.\mathsf{crsGen}(1^\lambda)$
    $\mathsf{key} \leftarrow \mathsf{SEE.KeyGen}(1^\lambda)$
    $K \leftarrow \mathsf{PRFKeyGen}(1^\lambda)$
    $\{P_i\}_{i\in[\ell]} \leftarrow \mathsf{Partition}(C)$
    Sample $r \leftarrow \{0,1\}^{M\ell}$
    For $i = 1$ to $\ell$ do:
      Let $C_{g,1}, C_{g,2}$ denote the two input gates of gate $g$
      $\widetilde{\mathsf{C}}^{\mathsf{eval}}_{i,k} \leftarrow \mathsf{GCircuit}(1^\lambda, \mathsf{C}^{\mathsf{real}}_{\mathsf{eval}}[\mathsf{crs}, C_{P_{i,k},1}, C_{P_{i,k},2}, P_{i,k}, (r_{C_{P_{i,k},1}}, r_{C_{P_{i,k},2}}, r_{P_{i,k}}),$
                                          $\mathsf{PRF}_K(1, i, k, 0), \mathsf{PRF}_K(1, i, k, 1)],$
                              $\{\mathsf{PRF}_K(0, i, j, b)\}_{j\in[\lambda], b\in\{0,1\}})$
```

```
**Circuit** $\mathsf{C}_{\mathsf{eval}}^{\tau \in \{\mathsf{real},\ \mathsf{ideal}\}}$
  **Hardwired Values**: $\mathsf{crs}, i, j, g, (r_i, r_j, r_g), \mathsf{lab}_0, \mathsf{lab}_1$
  **Input**: $\mathsf{d}$
  If $\tau$ is $\mathsf{real}$, define for all $\alpha, \beta \in \{0,1\}$, $\gamma(\alpha, \beta) := \mathsf{NAND}(\alpha \oplus r_i, \beta \oplus r_j) \oplus r_g$
  If $\tau$ is $\mathsf{ideal}$, define for all $\alpha, \beta \in \{0,1\}$, $\gamma(\alpha, \beta) := r_g$
  $f_b \leftarrow \mathsf{Send}\left(\mathsf{crs}, \mathsf{d}, j, (\gamma(b,0), \mathsf{lab}_{\gamma(b,0)}), (\gamma(b,1), \mathsf{lab}_{\gamma(b,1)})\right)$ for each $b \in \{0,1\}$
  Output $\mathsf{Send}(\mathsf{crs}, \mathsf{d}, i, f_0, f_1)$
```

Figure 8: Description of the Evaluation Circuit

```
      Let $\mathsf{keyEval} = \{\mathsf{PRF}_K(1, i, k, b)\}_{k \in [M], b \in \{0,1\}}$
      Let $\mathsf{keyHash} = \{\mathsf{PRF}_K(2, i, j, b)\}_{j \in [\lambda], b \in \{0,1\}}$
      $\{\tilde{\mathsf{C}}_{i,j}^{\mathsf{hash}}\}_{j \in [M-1]} \leftarrow \mathsf{GHash}(1^\lambda, \ell OT_{\mathsf{const}}.\mathsf{Hash}, \mathsf{keyEval}, \mathsf{keyHash})$
      Let $C_i^{\mathsf{write}} = \ell OT.\mathsf{SendWriteBlock}\left(\mathsf{crs}, \cdot, i, \{\mathsf{PRF}_K(0, i+1, j, b)\}_{j \in [\lambda], b \in \{0,1\}}\right)$
      $\tilde{\mathsf{C}}_i^{\mathsf{write}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, C_i^{\mathsf{write}}, \mathsf{keyHash}\right)$
    $c \leftarrow \mathsf{SEE.Enc}\left(\mathsf{key}, \left\{\tilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}, \tilde{\mathsf{C}}_{i,j}^{\mathsf{hash}}, \tilde{\mathsf{C}}_i^{\mathsf{write}}\right\}\right)$
    Output $\tilde{C} := (\mathsf{crs}, c, \{P_i\}_{i \in [\ell]})$ and $\mathsf{st} := (\mathsf{crs}, r, \mathsf{key}, \ell, K)$
```

- $\mathsf{AdaGInput}(\mathsf{st}, x)$:

```
    Parse $\mathsf{st} := (\mathsf{crs}, r, \mathsf{key}, \ell, K)$
    $D \leftarrow r_1 \oplus x_1 || ... || r_n \oplus x_n || 0^{M\ell - n}$
    $(\mathsf{d}, \hat{D}) \leftarrow \ell OT.\mathsf{Hash}(\mathsf{crs}, D)$
    Output $(\{\mathsf{PRF}_K(0, 1, j, \mathsf{d}_j)\}_{j \in [\lambda]}, r_1 \oplus x_1 || ... || r_n \oplus x_n, \mathsf{key}, r_{N-m+1} || ... || r_N)$
```

- $\mathsf{AdaEval}(\tilde{C}, \tilde{x})$:

```
    Parse $\tilde{C} := (\mathsf{crs}, c, \{P_i\}_{i \in [\ell]})$
    Parse $\tilde{x} := (\{\mathsf{lab}_{0,j}\}_{j \in [\lambda]}, s_1 || ... || s_n, \mathsf{key}, r_{N-m+1} || ... || r_N)$
    $D \leftarrow s_1 || ... || s_n || 0^{M\ell - n}$
    $(\mathsf{d}, \hat{D}) \leftarrow \ell OT.\mathsf{Hash}(\mathsf{crs}, D)$
    $\left\{\tilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}, \tilde{\mathsf{C}}_{i,j}^{\mathsf{hash}}, \tilde{\mathsf{C}}_i^{\mathsf{write}}\right\} \leftarrow \mathsf{SEE.Dec}(\mathsf{key}, c)$
    For $i = 1$ to $\ell$ do:
      Let $C_{g,1}, C_{g,2}$ denote the two input gates of gate $g$
      $\mathsf{e} \leftarrow \mathsf{GCEval}(\tilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}, \{\mathsf{lab}_{0,j}\}_{j \in [\lambda]})$
      $\mathsf{e} \leftarrow \ell OT.\mathsf{Receive}^{\hat{D}}(\mathsf{crs}, \mathsf{e}, C_{P_{i,k},1})$
      $(\gamma_k, \mathsf{lab}_{1,k}) \leftarrow \ell OT.\mathsf{Receive}^{\hat{D}}(\mathsf{crs}, \mathsf{e}, C_{P_{i,k},2})$
      $\{\mathsf{lab}_{2,j}\}_{j \in [\lambda]} \leftarrow \mathsf{GHEval}(\{\tilde{\mathsf{C}}_{i,j}^{\mathsf{hash}}\}_{j \in [M-1]}, \{\mathsf{lab}_{1,k}\}_{k \in [M]})$
      $\mathsf{e} \leftarrow \mathsf{GCEval}(\tilde{\mathsf{C}}_i^{\mathsf{write}}, \{\mathsf{lab}_{2,j}\}_{j \in [\lambda]})$
```

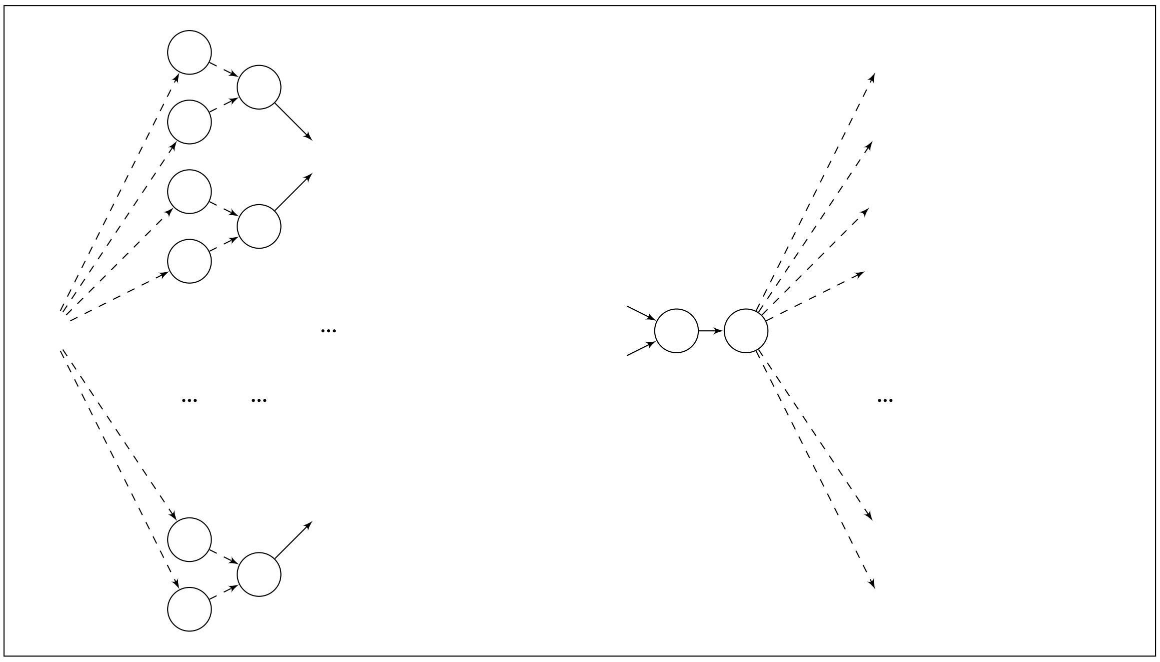

Figure 9: Illustration of the pebbling graph for one layer: $\tilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}$ are leaf nodes, $\tilde{\mathsf{C}}_{i,j}^{\mathsf{hash}}$ are intermediate nodes and the root node, finally $\tilde{\mathsf{C}}_{i}^{\mathsf{write}}$ is the extra node at the end. Dotted edges indicate where ℓOT is invoked. Note that $\mathsf{WriteBlock}$ is only invoked once and its result is reused M times.

```
    $\{\mathsf{lab}_{0,j}\}_{j\in[\lambda]} \leftarrow \ell OT.\mathsf{ReceiveWriteBlock}^{\tilde{\mathsf{D}}}(\mathsf{crs}, i, \{\gamma_k\}_{k\in[M]}, \mathsf{e})$
Recover the contents of the memory $D$ from the final state $\hat{D}$
Output $D_{N-m+1} \oplus r_{N-m+1}||...||D_N \oplus r_N$
```

Communcation Complexity of $\mathsf{AdaGInput}$. It follows from the construction that the communication complexity of $\mathsf{AdaGInput}$ is $\lambda^2 + n + m + |\mathsf{key}|$. From the parameters used in the somewhere equivocal encryption and the efficiency of block writing for laconic oblivious transfer, we note that $|\mathsf{key}| = \mathsf{poly}(\lambda, \log|C|)$.

Computational Complexity of $\mathsf{AdaGInput}$. The running time of $\mathsf{AdaGInput}$ grows linearly with $|C|$. However, it's possible to delegate the hashing of zeros to the offline phase, i.e. $\mathsf{AdaGCircuit}$. In that case, the running time only grows linearly with $n + \log|C|$.

Parallel Efficiency. With a good $\mathsf{Partition}$ algorithm and number of processors as many as the width of the circuit, $\mathsf{AdaEval}$ is able to run in $d \cdot \mathsf{poly}(\lambda, \log|C|)$ where d is the depth of the circuit.

Correctness. We note that for each wire (up to permutation due to rewiring), our construction manipulates the database and produces the final output the same way as the construction given by [GS18]. Therefore by the correctness of their construction, our construction outputs $C(x)$ with probability 1.

Adaptive Security. We formally prove the adaptive security in Appendix A.

5 Adaptive Garbled Parallel RAM

In this section, we will give a construction of adaptive garbled parallel RAM from public key assumptions.

Definition 5.1. *An adaptive garbled PRAM scheme* GPRAM *consists of the following PRAM programs.*

- $(\tilde{D}, SK) \leftarrow \mathsf{GPRAM.Memory}(1^\lambda, D)$*: It takes as input the security parameter* λ *and a database* D *as input and outputs a garbled database, and outputs a garbled database* $\tilde{D}$ *and a secret key* SK*;*
- $\tilde{P} \leftarrow \mathsf{GPRAM.Program}(SK, i, \Pi)$*: It takes as input the secret key* SK*, a sequence number* i *and a PRAM program* Π *as input, and outputs a garbled program* $\tilde{P}$*;*
- $\tilde{x} \leftarrow \mathsf{GPRAM.Input}(SK, i, x)$*: It takes as input the secret key* SK*, a sequence number* i *and input string* x*, and outputs the garbled input* $\tilde{x}$*;*
- $(y, \mathsf{st}') \leftarrow \mathsf{GPRAM.Eval}^{\tilde{D}}(\mathsf{st}, \tilde{\Pi}, \tilde{x})$*: It has random read-write access to* $\tilde{D}$*. It takes as input the state information* st *(empty for the first program), garbled program* $\tilde{\Pi}$*, and the garbled input* $\tilde{x}$*, outputs the evaluation result* y *and the updated state information*st'*, and updates the database* $\tilde{D}$ *accordingly.*

Correctness. *We require that for every database* $D, t = \mathsf{poly}(\lambda)$ *and every sequence of program* $\Pi_1, ..., \Pi_t$ *and their respective inputs* $x_1, ..., x_t$*, we have that*

$$\Pr[\mathsf{GPRAMCorrectExpt}(1^\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{GPRAMCorrectExpt}(1^\lambda)$ *is described in Figure 10.*

Adaptive Security. *We require that there exists stateful PPT simulators* ($\mathsf{SimMemory}$, $\mathsf{SimProgram}$, $\mathsf{SimInput}$)*, such that for all non-uniform PPT stateful adversary* $\mathcal{A}$ *and* $t = \mathsf{poly}(\lambda)$*, we have that:*

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

where $\mathsf{GPRAMSecExpt}^b(1^\lambda, \mathcal{A})$ *is described in Figure 11.*

```
$\mathsf{GPRAMCorrectExpt}(1^\lambda)$
  $(\tilde{D}, SK) \leftarrow \mathsf{Memory}(1^\lambda, D)$
  $\mathsf{st} \leftarrow \perp$
  For $i \in [t]$ do
    $\tilde{\Pi}_i \leftarrow \mathsf{Program}(SK, i, \Pi_i)$
    $\tilde{x}_i \leftarrow \mathsf{Input}(SK, i, x_i)$
    $y_i \leftarrow \Pi_i^D(x_i)$
    $(\tilde{y}_i, \mathsf{st}) \leftarrow \mathsf{Eval}^{\tilde{D}}(i, \mathsf{st}, \tilde{\Pi}_i, \tilde{x}_i)$
  Output 1 if there exists an $i \in [t]$ such that $\tilde{y}_i \neq y_i$.
```

Figure 10: GPRAM Correctness Experiment

```
$\mathsf{GPRAMSecExpt}^{b \in \{\mathsf{real,\ ideal}\}}(1^\lambda, \mathcal{A})$
  $(D, 1^n, 1^t) \leftarrow \mathcal{A}(1^\lambda)$
  If $\tau$ is real, $(\tilde{D}, SK) \leftarrow \mathsf{Memory}(1^\lambda, D)$
  If $\tau$ is ideal, $(\tilde{D}, SK) \leftarrow \mathsf{SimMemory}(1^\lambda, 1^|D|)$
  For $i \in [t]$ do
    $(\Pi_i, T_i) \leftarrow \mathcal{A}(\tilde{D}, \{\tilde{\Pi}_j, \tilde{x}_j\}_{j \in [i-1]})$
    If $\tau$ is real, $\tilde{\Pi}_i \leftarrow \mathsf{Program}(SK, i, \Pi_b)$
    If $\tau$ is ideal, $\tilde{\Pi}_i \leftarrow \mathsf{SimProgram}(1^{|T_i|})$
    $x_i \leftarrow \mathcal{A}(\mathsf{st}, \{\tilde{\Pi}_j, \tilde{x}_j\}_{j \in [i-1]}, \tilde{\Pi}_i)$
    If $\tau$ is real, $\tilde{x}_i \leftarrow \mathsf{Input}(SK, i, x_i)$
    If $\tau$ is ideal, $\tilde{\Pi}_i \leftarrow \mathsf{SimInput}(\Pi_i^D(x_i))$
  Output $\mathcal{A}(\mathsf{st}, \{\tilde{\Pi}_j, \tilde{x}_j\}_{j \in [t]})$ if $\Pi_i^D(x_i)$ runs in time $T_i$ for every $i \in [t]$
```

Figure 11: GPRAM Adaptive Security Game

Parallel Efficiency. *We require the following efficiency properties:*

- *The parallel running time of* Memory *should be bounded by* $|D|/M \cdot \mathsf{poly}(\log |D|, \lambda)$*, assuming* $|D| \geq M$*;*
- *The parallel running time of* Program *should be bounded by* $T \cdot \mathsf{poly}(\log M, \log |D|, n, \lambda)$*, where* T *is the parallel running time for the program;*
- *The running time of* Input *should be bounded by* $n \cdot \mathsf{poly}(\lambda, \log |D|, \log T, \log n, \log M)$*;*
- *The parallel running time of* Eval *should be bounded by* $T \cdot \mathsf{poly}(\log M, \log |D|, \log T, n, \lambda)$*, where* T *is the parallel running time for the program.*

Remark 5.2. *In order to simplify the presentation, here we simply use* n *as the size of local CPU state, thus the size of both the input and the output is* nM*. However, in our construction, we can also hardwire some of the initial CPU states by sending that part of the garbled input in* Program*, so that the size of the garbled input will only scale linearly with the length of the actual input instead of the entire local state space; we can also omit sending the labels for the part of CPU states that are not a part of the output, so the size of the garbled input will also only scale linearly with the length of the actual output.*

5.1 Garbling with Unprotected Memory Access

Following the general techniques used to construct garbled PRAM in previous works [BCP16, LO17, GOS18], we first construct an intermediate premitive with weaker security requirements called *adaptive garbled PRAM with unprotected memory acccess*. Informally, a garbled PRAM 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 evaluator. We mostly follow the definition given in [GOS18].

Definition 5.3. *An adaptive garbled PRAM scheme with unprotected memory access* UG-PRAM *consists of the following parallel PPT algorithms:*

- $(\tilde{D}, SK) \leftarrow \mathsf{Memory}(1^\lambda, D)$*: It takes as input the security parameter* λ*, a database* $D \in \{0,1\}^{2^d \cdot \lambda}$ *and outputs a garbled database* $\tilde{D}$ *and a secret key* SK*.*
- $\tilde{\Pi} \leftarrow \mathsf{Program}(SK, i, \Pi)$*: It takes as input a secret key* SK*, a sequence number* i*, and a PRAM program* Π *as input (represented as a set of CPU step circuits) and outputs a garbled program* $\tilde{\Pi}$*.*
- $\tilde{x} \leftarrow \mathsf{Input}(SK, i, \{x_k\}_{k\in[M]})$*: It takes as input a secret key* SK*, a sequence number* i*, and* M *input strings (which acts as its initial CPU state) for each processor* $\{x_k\}_{k\in[M]}$ *and outputs the garbled input* $\tilde{x}$*.*
- $(y, \mathsf{st}') \leftarrow \mathsf{Eval}^{\tilde{D}}(i, \mathsf{st}, \tilde{\Pi}, \tilde{x})$*: It is a PRAM program with random read-write access to* $\tilde{D}$*. It takes a sequence number* i*, the state information* st*, garbled program* $\tilde{\Pi}$*, garbled input* $\tilde{x}$*, outputs the evaluation result* y *and the updated state information* st'*, and updates the database accordingly.*

```
$\mathsf{UGPRAMCorrectExpt}(1^\lambda)$
    $(\tilde{D}, SK) \leftarrow \mathsf{Memory}(1^\lambda, D)$
    $\mathsf{st} \leftarrow \bot$
    For $i \in [t]$ do
        $\tilde{\Pi}_i \leftarrow \mathsf{Program}(SK, i, \Pi_i)$
        $\tilde{x}_i \leftarrow \mathsf{Input}(SK, i, x_i)$
        $y_i \leftarrow \Pi_i^D(x_i)$
        $(\tilde{y}_i, \mathsf{st}) \leftarrow \mathsf{Eval}^{\tilde{D}}(i, \mathsf{st}, \tilde{\Pi}_i, \tilde{x}_i)$
    Output 1 if there exists an $i \in [t]$ such that $\tilde{y}_i \neq y_i$.
```

Figure 12: UGPRAM Correctness Experiment

```
$\mathsf{UGPRAMSecExpt}(1^\lambda, \mathcal{A}, b)$
    $(D, 1^n, 1^t) \leftarrow \mathcal{A}(1^\lambda)$
    $(\tilde{D}, SK) \leftarrow \mathsf{Memory}(1^\lambda, D)$
    For $i \in [t]$ do
        $(\Pi_0, \Pi_1) \leftarrow \mathcal{A}(\tilde{D}, \{\tilde{\Pi}_j, \tilde{x}_j\}_{j\in[i-1]})$
        $\tilde{\Pi}_i \leftarrow \mathsf{Program}(SK, i, \Pi_b)$
        $x_i \leftarrow \mathcal{A}(\mathsf{st}, \{\tilde{\Pi}_j, \tilde{x}_j\}_{j\in[i-1]}, \tilde{\Pi}_i)$
        $\tilde{x}_i \leftarrow \mathsf{Input}(SK, i, x_i)$
    Output $\mathcal{A}(\mathsf{st}, \{\tilde{\Pi}_j, \tilde{x}_j\}_{j\in[t]})$ if the output of each ungarbled step circuit is the same for
all pairs of $(\Pi_0, \Pi_1)$
```

Figure 13: UGPRAM Adaptive Security Game

Correctness. *For every database D, $t = \mathsf{poly}(\lambda)$ and every sequence of program and input pairs $\{(P_i, x_i)\}_{i\in[t]}$, we have that*

$$\Pr[\mathsf{UGPRAMCorrectExpt}(1^\lambda) = 1] = 0,$$

where $\mathsf{UGPRAMCorrectExpt}(1^\lambda)$ *is described in Figure 12.*

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

$$|\Pr[\mathsf{UGPRAMSecExpt}(1^\lambda, \mathcal{A}, 0) = 1] - \Pr[\mathsf{UGPRAMSecExpt}(1^\lambda, \mathcal{A}, 1) = 1]| \leq \mathsf{negl}(\lambda),$$

where $\mathsf{UGPRAMSecExpt}(1^\lambda, \mathcal{A}, b)$ *is described in Figure 13.*

Parallel Efficiency. *We require the following efficiency properties:*

- *The parallel running time of* Memory *should be bounded by $|D|/M \cdot \mathsf{poly}(\log|D|, \lambda)$, assuming $|D| \geq M$;*

- *The parallel running time of* $\mathsf{Program}$ *should be bounded by* $T \cdot \mathsf{poly}(\log M, \log |D|, n, \lambda)$*, where* T *is the parallel running time for the program;*

- *The running time of* Input *should be bounded by* $n \cdot \mathsf{poly}(\lambda, \log |D|, \log T, \log n, \log M)$*;*

- *The parallel running time of* Eval *should be bounded by* $T \cdot \mathsf{poly}(\log M, \log |D|, \log T, n, \lambda)$*, where* T *is the parallel running time for the program.*

5.2 Inter-CPU Communications for PRAM Programs

Recall from Section 2.2 that in our construction, we also need to allow PRAM programs to have inter-CPU communications (communications without using the memory) for parallel writes. We formalize such inter-CPU communication with one CPU reading state information from another CPU. Say CPU_i needs to read from CPU_j, this can be formalized as evaluating the step circuit $C^{\Pi}_{\mathsf{CPU}_i}(\mathsf{state}_i, \mathsf{state}_j) = \mathsf{state}'_i$, which takes as input the old states of two CPUs $\mathsf{state}_i, \mathsf{state}_j$ and outputs the updated states state'_i. This formalization enables two CPUs communicating with each other as well as broadcasts. Say CPU_j wants to transmit a message to CPU_i, it can prepare its message inside state_j, and in the next time-step CPU_i can read the message from the state_j and update its own state.

Here, we assume that the program that we are given is in the standard PRAM model, i.e. it does not use inter-CPU communication at all (this would not impact their computational power as they can use RAM as a medium to communicate). Similarly, inter-CPU communications in our construction can also be emulated by the standard PRAM model with constant overhead, but using inter-CPU communications enable us to more easily distinguish the memory accesses that do not need extra protections. Looking ahead, as we only need to invoke inter-CPU communications in parallel writes, security still holds since the communication patterns are fixed and do not leak any information about the database, the program nor the input.

5.3 Roadmap for UMA Security

To construct a GPRAM, we first construct a GPRAM with unprotected memory access (UGPRAM), which we later bootstrap to full security using techniques from [GOS18]. We will handle the additional subtleties in the full security case in Section 5.6. For now, we focus our attention to constructing UGPRAM.

The main idea is to expand our weak garbling scheme from Section 4 to a full garbling scheme for PRAM programs with unprotected memory access. The main difficulty here is that in PRAM programs since the memory location we access will depend on the input, we cannot simply rearrange the memory access into a single continuous block. Thus, a more general parallel write for ℓOT is required. We will highlight some of the changes we made to ℓOT in this section, with the full construction presented in Section 5.4.

Recall Figure 1 for the transformation toy example. In the ungarbled world, there are M CPU step circuits for each parallel time-step, which naturally corresponds to M Yao's garbled circuits in the garbled world. Since we need to use inter-CPU communications in our update phase (there are two outgoing edges in the transformed graph), the garbled step circuit for CPU i needs to hardwire not only the input label for the next step circuit for

CPU i, but also labels for message sent to CPU j. This introduces intricate simulation dependencies that we do not know how to pebble efficiently.

In order to tackle this issue, we employ the technique of "parallel checkpoints," similar to branch and combine transformation from [CCC+16], which is a technique that allow us to compile arbitrary PRAM programs such that the compiled programs have good pebbling complexity. On a high level, after every CPU has evaluated a time-step on its own, instead of every CPU routes its messages (which we simply formalize as CPU states) on its own, we first aggregate all the CPU states into a single garbled circuit that acts like a checkpoint, and the garbled step circuits in the next time-step takes their input labels directly from this single garbled circuit. With this change, we break up the intricate simulation dependencies by aggregating all the simulation dependencies into this single circuit, and in order to pebble circuits in the next parallel time-step, it is sufficient to pebble this single checkpoint.

However, the size of all CPU states is $\Omega(M)$, so the size of the checkpoint is too large. To fix this, instead of simply aggregating the states, we use a garbled Merkle tree from Section 4.3 (for which there is a good pebbling strategy) that evaluates the ℓOT hash for an ephemeral database that holds all CPU states. Since the digest binds the entire database (in this case, all CPU states), this allows us to compress the ephemeral database into a short string and all the circuits in the garbled Merkle tree is small. Overall, we maintain the linear structure in the simulation dependency graph.

Finally, in order to keep the write ciphertext small so that all the intermediate circuit is of small size, here we break the parallel write into three procedures:

1. (inside garbled PRAM) A send phase where each processor $k \in [M]$ decides a location L_k to write to independently, and generates a parallel write ciphertext e_k, which is a sequence of garbled circuits that reads out all the labels for the neighbouring digests;
2. (outside garbled PRAM) A receive phase where the receiver can evaluate the parallel write ciphertexts in parallel, and return all the labels for the neighbouring digests;
3. (inside garbled PRAM) Finally an update phase where processors on receiving the labels, jointly computes the updated digest in parallel using inter-CPU communications.

With these three parallel write procedures, we wrap all the memory access in the original PRAM program with ℓOT procedures to access the memory.

5.4 Parallel Updatable Laconic OT

Recall that ℓOT already allows for parallel reads. In this section, we will formalize the syntax for parallel writes.

Definition 5.4 (Parallel Updatable ℓOT). *A parallel updatable ℓOT scheme consists of algorithms* crsGen, Hash, Send, $\mathsf{Receive}$ *as in Definition 3.4 and additionally three algorithms* $\mathsf{SendPWrite}$, $\mathsf{ReceivePWrite}$, *and* $\mathsf{UpdatePWrite}$ *with the following syntax.*

- $\mathsf{e}_k \leftarrow \mathsf{SendPWrite}\left(\mathsf{crs}, \mathsf{digest}, L_k, \{m_{k,i,j,c}\}_{i\in[d],j\in[2\lambda],c\in\{0,1\}}\right)^4$. *It takes as input the common reference string* crs, *a digest* digest, *one location* L_k, *and some pairs of*

[4] As we will see later, $\mathsf{SendPWrite}$ only prepares the ciphertext for preparing the write, so b_k is not needed.

messages $\{m_{k,i,j,c}\}_{i\in[d],j\in[2\lambda],c\in\{0,1\}}$*, where each* $m_{k,i,j,c}$ *is of length* λ*. We require that each* L_k *is different. It outputs a ciphertext* e_k*.*

- $\mathsf{trace} \leftarrow \mathsf{ReceivePWrite}^{\tilde{\mathsf{D}}}(\mathsf{crs}, \{L_k\}_{k\in[M]}, \{b_k\}_{k\in[M]}, \{\mathsf{e}_k\}_{k\in[M]})$*. This is a PRAM algorithm with random read/write access to* $\tilde{\mathsf{D}}$*. It takes as input the common reference string* crs*,* M *locations* $\{L_k\}_{k\in[M]}$ *and bits to be written* $\{b_k\}_{k\in[M]}$*, and* M *ciphertexts* $\{\mathsf{e}_k\}_{k\in[M]}$*. It updates the state* $\tilde{\mathsf{D}}$ *(such that* $D[L_k] = b_k$ *for every* $k \in [M]$*) and outputs its computation trace* trace*.*

- $\mathsf{digest}^* \leftarrow \mathsf{UpdatePWrite}(\mathsf{crs}, \{L_k\}_k, \{b_k\}_k, \{m_{k,i,j,c}\}_{i,j,c}, \mathsf{trace})$*. This is a PRAM algorithm that takes as input the common reference string* crs*, a digest* digest*,* M *locations* $\{L_k\}_{k\in[M]}$*,* M *bits* $\{b_k\}_{k\in[M]}$*, the same pairs of messages* $\{m_{k,i,j,c}\}_{i\in[d],j\in[2\lambda],c\in\{0,1\}}$ *as used in* $\mathsf{SendPWrite}$*, and the trace* trace *computed by* $\mathsf{ReceivePWrite}$*. It outputs the new digest* digest^* *using only inter-CPU communications.*

On a high level, for the parallel version of SendWrite and ReceiveWrite, we basically do the same except that we need to send back all the digest since the computation of the updated digest now happens in the UpdatePWrite PRAM program, where all processors collaboratively update the partial Merkle tree in the input and compute the updated root hash. This is a clean way to keep each pieces in SendPWrite independent from each other. Looking ahead, this change will help us to use this construction in a black-box way without compromising online complexity.

Formally, similarly to updatable ℓOT, we require the following properties for the parallel write procedures:

- **Correctness of Parallel Writes:** Let database D be of size at most $\mathsf{poly}(\lambda)$ and let $\{L_k\}_{k\in[M]}$ be M non-overlapping memory locations. Let D^* be a database that is identical to D except that $D^*[L_k] = b_k$ for every $k \in [M]$. For any sequence of messages $\{m_{j,c}\}_{j\in[2\lambda],c\in\{0,1\}}$, we require that

$$\Pr\left[\mathsf{d}' = \mathsf{d}^* \left| \begin{array}{r l} \mathsf{crs} & \leftarrow \mathsf{crsGen}(1^\lambda) \\ (\mathsf{d}, \tilde{\mathsf{D}}) & \leftarrow \mathsf{Hash}(\mathsf{crs}, D) \\ (\mathsf{d}^*, \tilde{\mathsf{D}}^*) & \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*) \\ \mathsf{e}_k & \leftarrow \mathsf{SendPWrite}\left(\mathsf{crs}, \mathsf{d}, L_k, \{m_{k,i,j,c}\}_{i,j,c}\right) \forall k \\ \mathsf{trace} & \leftarrow \mathsf{ReceivePWrite}^{\tilde{\mathsf{D}}}(\mathsf{crs}, \{L_k\}_k, \{b_k\}_k, \{\mathsf{e}_k\}_k) \\ \mathsf{d}' & \leftarrow \mathsf{UpdatePWrite}(\mathsf{crs}, \{L_k\}_k, \{b_k\}_k, \{m_{k,i,j,c}\}_{i,j,c}, \mathsf{trace}) \end{array}\right.\right] = 1,$$

- **Sender Privacy Against Semi-Honest Receivers With Regard To Parallel Writes:** There exists a PPT simulator $\ell\mathsf{OTSim}$ such that for any non-uniform PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ there exists a negligible function $\mathsf{negl}(\cdot)$ s.t.

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

where $\mathsf{PWriSenPriv}^{\mathsf{real}}$ and $\mathsf{PWriSenPriv}^{\mathsf{ideal}}$ are described in Figure 14.

```
$\mathsf{PWriSenPriv}^{\tau\in\{\mathsf{real,\ ideal}\}}(1^\lambda, \mathcal{A})$
    $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$
    $(D, L_k, \{m_{k,i,j,c}\}_{i\in[d],j\in[\lambda],c\in\{0,1\}}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{crs})$
    $(d, \hat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
    Let $\hat{\mathsf{D}}_{i,j}$ be the $j$-th bit of the digest on the $i$-th item in path $L_k$
    If $\tau$ is $\mathsf{real}$
        $\mathsf{e}_k \leftarrow \mathsf{SendPWrite}\left(\mathsf{crs}, \mathsf{d}, L_k, \{m_{k,i,j,c}\}_{i\in[d],j\in[2\lambda],c\in\{0,1\}}\right)$
    If $\tau$ is $\mathsf{ideal}$
        $\mathsf{e}_k \leftarrow \ell\mathsf{OTSimPWrite}\left(\mathsf{crs}, D, L_k, \{m_{k,i,j,\hat{\mathsf{D}}_{i,j}}\}_{i\in[d],j\in[2\lambda]}\right)$
    Output $\mathcal{A}_2(\mathsf{st}, \mathsf{e}_k)$
```

Figure 14: Sender Privacy for Parallel Writes Security Game

```
**Circuit** $\mathsf{C}^{\mathsf{trav}}$
    **Hardwired Values**: $\mathsf{crs}, b, \mathsf{Keys}, \widetilde{\mathsf{Keys}}, r, \tilde{r}$
    **Input**: $\mathsf{sbl}$
    Parse $\mathsf{sbl} = (\mathsf{sbl}_0, \mathsf{sbl}_1)$
    $\mathsf{e} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}_b, \mathsf{Keys}; r)$
    $\tilde{\mathsf{e}} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}_b, \widetilde{\mathsf{Keys}}; \tilde{r})$
    Output $(\mathsf{e}, \tilde{\mathsf{e}})$
```

Figure 15: The Traversing Circuit $\mathsf{C}^{\mathsf{trav}}\left[\mathsf{crs}, b, \mathsf{Keys}, \widetilde{\mathsf{Keys}}, r, \tilde{r}\right]$ from [CDG+17]

- **Efficiency Requirements:** We require that all three algorithms run in time $\mathsf{poly}(\log|D|, \lambda, \log M)$ on a PRAM machine with M processors using only local inter-CPU communications.

Theorem 5.5. *Assuming the existence of laconic oblivious transfer, there exists laconic oblivious transfer with parallel writes.*

We now give the formal construction of the three new algorithms in parallel updatable ℓOT. In the construction, for each $k \in [M]$ is implied.

- $\mathsf{crsGen}(1^\lambda)$, $\mathsf{Hash}(\mathsf{crs}, D)$, $\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$ and $\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L)$ is constructed in the same way as in updatable ℓOT given in [CDG+17].

- $\mathsf{SendPWrite}\left(\mathsf{crs}, \mathsf{digest}, L_k, \{m_{k,i,j,c}\}_{i\in[d],j\in[2\lambda],c\in\{0,1\}}\right)$:

```
    Parse $L_k = (b_{k,1}, b_{k,2}, ..., b_{k,d-1}, t_k)$
    Let $\mathsf{Keys}_k^d$ be $0^*$
    Let $\widetilde{\mathsf{Keys}}_k^i$ to be $\{m_{k,i,j,c}\}_{j\in[2\lambda],c\in\{0,1\}}$
    For $i = d-1$ downto 1:
        Pick $\mathsf{Keys}_k^i$ as input keys for $\mathsf{C}^{\mathsf{trav}}$
        Pick $r_k^i, \tilde{r}_k^i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$
        $\tilde{\mathsf{C}}_k^i \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{trav}}\left[\mathsf{crs}, b_{k,i}, \mathsf{Keys}_k^{i+1}, \widetilde{\mathsf{Keys}}_k^{i+1}, r_k^i, \tilde{r}_k^i\right], \mathsf{Keys}_k^i\right)$
    $\mathsf{e}_k^0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \mathsf{Keys}_k^1)$
    $\tilde{\mathsf{e}}_k^0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \widetilde{\mathsf{Keys}}_k^1)$
    Output $\mathsf{e}_k = \left(\mathsf{e}_k^0, \tilde{\mathsf{e}}_k^0, \{\tilde{\mathsf{C}}_k^i\}_{i\in[d-1]}\right)$
```

- $\mathsf{ReceivePWrite}^{\hat{\mathsf{D}}}(\mathsf{crs}, \{L_k\}_{k\in[M]}, \{b_k\}_{k\in[M]}, \{\mathsf{e}_k\}_{k\in[M]})$:

```
    Parse $\mathsf{e}_k = \left(\mathsf{e}_k^0, \tilde{\mathsf{e}}_k^0, \{\tilde{\mathsf{C}}_k^i\}_{i\in[d-1]}\right)$
    Parse $L_k = (b_{k,1}, b_{k,2}, ..., b_{k,d-1}, t_k)$
    Parse $\hat{\mathsf{D}}$ as a Merkle tree
    Denote the end node of path $b_1 b_2 ... b_i$ by $\hat{\mathsf{D}}_{b_1 b_2 ... b_i}$.
    For $i = 1$ to $d-1$:
        $\mathsf{sbl}_k^i \leftarrow (\hat{\mathsf{D}}_{b_{k,1}...b_{k,i-1}0}, \hat{\mathsf{D}}_{b_{k,1}...b_{k,i-1}1})$
        $\mathsf{Labels}_k^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \mathsf{e}_k^{i-1}, \mathsf{sbl}_k^i)$
        $\widetilde{\mathsf{Labels}}_k^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \tilde{\mathsf{e}}_k^{i-1}, \mathsf{sbl}_k^i)$
        $(\mathsf{e}_k^i, \tilde{\mathsf{e}}_k^i) \leftarrow \mathsf{GCEval}\left(\tilde{\mathsf{C}}_i^k, \left\{\mathsf{Labels}_k^i\right\}_{k\in[M]}^{i\in[d]}\right)$
    $\mathsf{leaf}_k \leftarrow (\hat{\mathsf{D}}_{b_{k,1}...b_{k,d-1}0}, \hat{\mathsf{D}}_{b_{k,1}...b_{k,d-1}1})$
    $\widetilde{\mathsf{Labels}}_k^d \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \tilde{\mathsf{e}}_k^{d-1}, \mathsf{leaf})$
    Update the Merkle tree as in $\mathsf{UpdatePWrite}$
    Output $\mathsf{trace} = \left\{\widetilde{\mathsf{Labels}}_k^i\right\}_{k\in[M]}^{i\in[d]}$
```

- $\mathsf{UpdatePWrite}(\mathsf{crs}, \{L_k\}_{k\in[M]}, \{b_k\}_{k\in[M]}, \{m_{k,i,j,c}\}_{i\in[d],j\in[2\lambda],c\in\{0,1\}}, \mathsf{trace})$:

```
    Parse $\mathsf{trace} = \left\{\widetilde{\mathsf{Labels}}_k^i\right\}_{k\in[M]}^{i\in[d]}$
    Recover $(\mathsf{sbl}_k^1, ..., \mathsf{sbl}_k^{d-1}, \mathsf{leaf}_k)_{k\in[M]}$ from $\mathsf{trace}$ with $\{m_{k,i,j,c}\}_{i,j,c}$ as keys
    Parse $L_k = (b_{k,1}, b_{k,2}, ..., b_{k,d-1}, t_k)$
    Parse $\mathsf{sbl}_k^i = (\mathsf{sbl}_k^{i,0}, \mathsf{sbl}_k^{i,1})$ for $i \in [d-1]$
    $\mathsf{leaf}_k[t_k] \leftarrow b_k$
    $\mathsf{sbl}_k^d \leftarrow \mathsf{leaf}_k$
    For $i = d-1$ downto 1:
        Sort by $\{L_k\}$ and move inactive processors to the end
        $\mathsf{mark}_k \leftarrow 0$
        If neighboring processor's $\mathsf{path}$ is the same up to $i$
            Mark the processor with larger $k$ inactive and "terminate" it
            Update $\mathsf{sbl}_k^{i,1}$ with the value from the neighboring processor
        $\mathsf{sbl}_k^{i,b_{k,i}} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \mathsf{sbl}_k^{i+1})$
    Output $\mathsf{digest}^* = \ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \mathsf{sbl}_1^1)$
```

Here for the sake of presentation, we assume that each L_k is not only different, but also belongs to different blocks in the Merkle tree. However, it is very easy to extend the construction to handle writing to the same block and we will only incur an extra factor of $O(\min\{\log M, \lambda\})$ in runtime.

Efficiency. Send and $\mathsf{Receive}$ can be run independently, therefore its parallel run time is $\mathsf{poly}(\log|D|, \lambda)$. It can be seen that both the running time of $\mathsf{SendPWrite}$ and the parallel running time of $\mathsf{ReceivePWrite}$ is $\mathsf{poly}(\log|D|, \lambda, \log M)$, and the latter only uses local inter-CPU communications.

Correctness. Correctness with regard to parallel writes can easily be argued by the correctness of the garbling scheme and the correctness of ℓOT_{const}.

Security. For proof of security, see Appendix B.1.

5.5 Construction of UGPRAM

In this subsection, we give a construction of adaptive garbled PRAM with unprotected memory access. Formally,

Theorem 5.6. *Assuming the existence of parallel updatable laconic oblivious transfer, somewhere equivocal encryption, a pseudorandom function and garbling scheme for circuits with selective security, there exists adaptive garbled PRAM with unprotected memory access, where the time required to garble a database, a PRAM program and an input grows linearly (up to polylogarithmic factors) with the size of the database, parallel running time of the PRAM program and length of the input respectively.*

First, we give an overview of the construction. Similar to how we garble the circuit PRAM program, here we simply protect the database with ℓOT as we do not need to protect the memory (yet). We preprocess the PRAM program by appending the instruction $\ell OT.\mathsf{UpdatePWrite}$ after each parallel write operation, and then broadcasting the computed updated digest to all processors for the next instruction. For each PRAM timestep, each processor evaluates a selectively secure garbled step circuit, taking labels for the current database digest d, local CPU memory state (which consists of two local CPU memories) and possibly some data read according to the previous request rData as input; outputs a ℓOT read (or parallel write) ciphertext, and updates local CPU memory and database digest accordingly. Furthermore, each processor outputs two copies of the same updated local CPU memory, one stored in the clear state', one as labels for the input to a garbled Merkle tree. Using these two copies, the evaluator can put all the local memory together into a new database D' (the checkpoint), as well as evaluating the garbled Merkle tree to obtain labels d' for D'. After that, each processor evaluates another selectively secure garbled circuit to read out labels for the next garbled step circuit from D', after which D' can be safely discarded and replaced by the new database in the next iteration. The garbled program is simply the somewhere equivocal encryption of all the garbled circuits, and the garbled input consists of the decryption key and the input labels for the first M garbled step circuits.

Now we give the formal description of the construction. We use a somewhere equivocal encryption with block length set to $|\tilde{\mathsf{C}}|$, where $|\tilde{\mathsf{C}}|$ denotes the largest garbled circuit in the

```
**Step Circuit** $\mathsf{C}^{\mathsf{eval}}$
  **Hardwired Values/Circuit**: $\mathsf{crs}, \tau, \{\mathsf{lab}_{j,b}\}_{j\in[\lambda],b\in\{0,1\}}, C$
  **Input**: $\mathsf{d}, \mathsf{state}, \mathsf{rData}$
  $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) \leftarrow C^{\Pi}_{\mathsf{CPU}_i}(\mathsf{state}, \mathsf{rData})$
  If $\tau = T$, reset $\mathsf{lab}_{j+\lambda,b} = b$ for all $j \in [n]$
  If $\tau$ is at the end of $\ell OT.\mathsf{UpdatePWrite}$, $\mathsf{d} \leftarrow \mathsf{tr}_k$
  If $\mathsf{R/W} = \mathsf{write}$
    $e \leftarrow \ell OT.\mathsf{SendPWrite}(\mathsf{crs}, \mathsf{d}, L, \{b||\mathsf{lab}_{j+\lambda+n,b\oplus r_{n+j}}\}_{j\in[2d\lambda],b\in\{0,1\}})$
  Else if $\mathsf{R/W} = \mathsf{read}$
    $e \leftarrow \ell OT.\mathsf{Send}(\mathsf{crs}, \mathsf{d}, L, \{b||\mathsf{lab}_{j+\lambda+n,b\oplus r_{n+j}}\}_{j\in[2\lambda],b\in\{0,1\}})$
  Else $e \leftarrow \perp$
  Output $(\mathsf{R/W}, L, e, \mathsf{wData}, \mathsf{state}', \{\mathsf{lab}_{j,\mathsf{d}_j}\}_{j\in[\lambda]}, \{\mathsf{lab}_{j+\lambda,\mathsf{state}'_j}\}_{j\in[n]})$
```

Figure 16: Description of the Step (evaluation) Circuit

```
**Step Circuit** $\mathsf{C}^{\mathsf{read}}$
  **Hardwired Values/Circuit**: $\mathsf{crs}, \{L_j\}_{j\in[2n+2d\lambda]}, \{\mathsf{lab}_{j,b}\}_{j\in[n+2d\lambda],b\in\{0,1\}}$
  **Input**: $\mathsf{d}'$
  Output $\ell OT.\mathsf{Send}(\mathsf{crs}, \mathsf{d}', \{L_j\}_j, \{\mathsf{lab}_{j,b}\}_{j,b})$
```

Figure 17: Description of the Read Circuit

construction; the message length to be T (which is the running time of the preprocessed version of program $\hat{\Pi}$) and the equivocation parameter to be $\log T + \log n + \log M + O(1)$.

- $\mathsf{Memory}(1^\lambda, D)$:
  $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda, 1^{|D|})$
  $K \leftarrow \mathsf{PRFKeyGen}(1^\lambda)$
  $\mathsf{lab}^1_{j,b} \leftarrow \mathsf{PRF}_K(1, j, 0, b), \forall j \in [\lambda], b \in \{0,1\}$
  $(\mathsf{d}, \hat{D}) \leftarrow \ell OT.\mathsf{Hash}(\mathsf{crs}, D)$
  Output $\hat{D}, \{\mathsf{lab}^1_{j,\mathsf{d}_j}\}_{j\in[\lambda]}$ as the garbled memory and $SK = (K, \mathsf{crs}, |D|)$

- $\mathsf{Program}(SK, i, \Pi)$:
  Parse $SK = (K, \mathsf{crs}, |D|)$
  Expand every processor's local memory for temporary storage used by $\ell OT.\mathsf{UpdatePWrite}$, this region will be referred to as tr_k for CPU k
  Preprocess the program Π into $\hat{\Pi}$ by inserting instructions after every parallel write in $\hat{\Pi}$:
    $\mathsf{tr}_1 \leftarrow \ell OT.\mathsf{UpdatePWrite}(\mathsf{crs}, \{L_k\}_{k\in[M]}, \{b_k\}_{k\in[M]}, \{m_{k,i,j,c}\}_{i\in[d],j\in[\lambda],c\in\{0,1\}}, \{\mathsf{rData}_k\}_k)$
    $\mathsf{tr}_k \leftarrow \mathsf{tr}_1 \forall k \in [M]$
  Let T be the parallel execution time of $\hat{\Pi}$

```
  Sample $\mathsf{lab}^{\tau,k}_{j,b}, \overline{\mathsf{lab}}^{\tau,k}_{j,b}$ for $j \in [\lambda + n + 2d\lambda], b \in \{0,1\}, \tau \in [2,T], k \in [M]$ u.a.r.
  $\mathsf{lab}^{1,k}_{j,b} \leftarrow \mathsf{PRF}_K(i,j,0,b)$ for $j \in [\lambda + n + 2d\lambda], b \in \{0,1\}, k \in [M]$
  $\mathsf{lab}^{T+1}_{j,b} \leftarrow \mathsf{PRF}_K(i+1,j,k,b)$ for $j \in [\lambda + n + 2d\lambda], b \in \{0,1\}, k \in [M]$
  Let $\{\mathsf{lab}^{\tau,k}_{j,b}\}$ denote $\{\mathsf{lab}^{\tau,k}_{j,b}\}_{j \in [\lambda+n+2d\lambda], b \in \{0,1\}}$
  For $\tau = 1$ to $T$ do
    Let $\{C_k\}_{k \in [M]}$ be the parallel step circuits at time step $\tau$, $\{L_{k,j}\}_{k \in [M], j \in [2n+2d\lambda]}$ be the input locations of the step circuits at time step $\tau + 1$ (for circuits without inter-CPU communications, the second read is performed at arbitrary location)
    $\widetilde{\mathsf{C}}^{\mathsf{eval}}_{\tau,k} \leftarrow \mathsf{GCircuit}(1^\lambda, \mathsf{C}^{\mathsf{eval}}[\mathsf{crs}, \tau, \{\overline{\mathsf{lab}}^{\tau,k}_{j,b}\}, C_k], \{\mathsf{lab}^{\tau,k}_{j,b}\})$ for every $C_k$
    Let $\mathsf{keyEval} = \{\mathsf{lab}^{\tau,k}_{\lambda+j,b}\}_{j \in [n+2d\lambda], b \in \{0,1\}, k \in [M]}$
    Let $\mathsf{keyHash} = \{\mathsf{lab}^{\tau,k}_{j,b}\}_{j \in [\lambda], b \in \{0,1\}, k \in [M]}$
    $\{\widetilde{\mathsf{C}}^{\mathsf{hash}}_{\tau,j}\}_{j \in [(n+2d\lambda)M-1]} \leftarrow \mathsf{GHash}(1^\lambda, \ell OT_{\mathsf{const}}.\mathsf{Hash}, \mathsf{keyEval}, \mathsf{keyHash})$
    $\widetilde{\mathsf{C}}^{\mathsf{read}}_{\tau,k} \leftarrow \mathsf{GCircuit}(1^\lambda, \mathsf{C}^{\mathsf{read}}[\mathsf{crs}, \{L_{k,j}\}_j, \{\mathsf{lab}^{\tau+1,k}_{j,b}\}], \{\overline{\mathsf{lab}}^{\tau,k}_{j,b}\})$ for every $C_k$
  $\mathsf{key} \leftarrow \mathsf{SEE.KeyGen}(1^\lambda; \mathsf{PRF}_K(i,0,0,0))$
  Output $\tilde{\Pi} := \mathsf{SEE.Enc}(\mathsf{key}, \{\widetilde{\mathsf{C}}^{\mathsf{eval}}_{\tau,k}, \widetilde{\mathsf{C}}^{\mathsf{hash}}_{\tau,j}, \widetilde{\mathsf{C}}^{\mathsf{read}}_{\tau,k}\}_{\tau,j,k})$
```

- $\mathsf{Input}(SK, i, \{x_k\}_{k \in [M]})$:

```
  Parse $SK = (K, \mathsf{crs}, |D|)$
  $\mathsf{lab}^{1,k}_{j,b} \leftarrow \mathsf{PRF}_K(1,j,k,b)$ for $j \in [\lambda + n + 2d\lambda], b \in \{0,1\}, k \in [M]$
  $\mathsf{lab}^{T+1,k}_{j,b} \leftarrow \mathsf{PRF}_K(T+1,j,k,b)$ for $j \in [\lambda + n + 2d\lambda], b \in \{0,1\}, k \in [M]$
  $\mathsf{key} \leftarrow \mathsf{SEE.KeyGen}(1^\lambda; \mathsf{PRF}_K(i,0,0,0))$
  Output $(\mathsf{key}, \{\mathsf{lab}^{1,k}_{\lambda+j,x_{k,\lambda+j}}\}_{j \in [n], k \in [M]}, \{\mathsf{lab}^{T+1,k}_{\lambda+j,x_{k,\lambda+j}}\}_{j \in [n], k \in [M]}, \{\mathsf{lab}^{1,k}_{\lambda+n+j,0}\}_{j \in [2d\lambda], k \in [M]})$
```

- $\mathsf{Eval}^{\tilde{D}}(i, \mathsf{st}, \tilde{\Pi}, \tilde{x})$:

```
  Parse $\tilde{x} = (\mathsf{key}, \{\mathsf{lab}^{1,k}_{\lambda+j,x_{k,\lambda+j}}\}_{j \in [n]}, \mathsf{outlab}, \{\mathsf{lab}^{1,k}_{\lambda+n+j,0}\}_{j \in [2d\lambda]})$
  If $i = 1$, obtain $\{\mathsf{lab}_j\}_{j \in [\lambda]}$ from $\tilde{D}$ and for every $k \in [M]$ let $\{\mathsf{lab}_{j,k}\}_{j \in [\lambda]} \leftarrow \{\mathsf{lab}_j\}_{j \in [\lambda]}$,
  else parse $\mathsf{st}$ as $\{\mathsf{lab}_{j,k}\}_{j \in [\lambda], k \in [M]}$
  $\{\widetilde{\mathsf{C}}^{\mathsf{eval}}_{\tau,k}, \widetilde{\mathsf{C}}^{\mathsf{hash}}_{\tau,j}, \widetilde{\mathsf{C}}^{\mathsf{read}}_{\tau,k}\}_{\tau,j,k} \leftarrow \mathsf{SEE.Dec}(\mathsf{key}, \tilde{Pi})$
  $\overline{\mathsf{lab}}_k \leftarrow \{\mathsf{lab}_{j,k}\}_{j \in [\lambda+n+2d\lambda]}$
  For $\tau = 1$ to $T$ do
    $(\mathsf{R/W}, L_k, e_k, \mathsf{wData}_k, \mathsf{state}'_k, \{\mathsf{lab}_j\}_{j \in [\lambda]}, \overline{\mathsf{lab}}) \leftarrow \mathsf{GEval}(\widetilde{\mathsf{C}}^{\mathsf{eval}}_{\tau,k}, \overline{\mathsf{lab}})$ for each $\widetilde{\mathsf{C}}^{\mathsf{eval}}_{\tau,k}$
    If $\mathsf{R/W} = \mathsf{write}$,
      $\{\mathsf{rData}_{j,k} || \mathsf{lab}_{\lambda+n+j,k}\}_{j \in [2d\lambda],k} \leftarrow \ell OT.\mathsf{ReceivePWrite}^{\tilde{\mathsf{D}}}(\mathsf{crs}, \{L_k\}_k, \{\mathsf{wData}_k\}_k, \{\mathsf{e}_k\}_k)$
    Else if $\mathsf{R/W} = \mathsf{read}$,
      $\{\mathsf{rData}_{j,k} || \mathsf{lab}_{\lambda+n+j,k}\}_{j \in [2\lambda]} \leftarrow \ell OT.\mathsf{Receive}^{\tilde{\mathsf{D}}}(\mathsf{crs}, L, \mathsf{e}_k)$ for every $k \in [M]$
    $(\tilde{D}', \mathsf{d}') \leftarrow \ell OT.\mathsf{Hash}(\mathsf{crs}, D')$ where $D' = \{\mathsf{state}'_k, \mathsf{rData}_{j,k}\}_{k \in [M], j \in [2d\lambda]}$
    $\tilde{\mathsf{d}}' \leftarrow \mathsf{GHEval}(\{\widetilde{\mathsf{C}}^{\mathsf{hash}}_{\tau,j}\}_j, \{\mathsf{lab}_j\}_j)$
    $\overline{\mathsf{lab}}_k \leftarrow \{\mathsf{lab}_{j,k}\}_{j \in [\lambda]} + \ell OT.\mathsf{Receive}(\mathsf{GEval}(\widetilde{\mathsf{C}}^{\mathsf{read}}, \tilde{\mathsf{d}}'))$
```

Decrypt output y from $\{\mathsf{lab}_{\lambda+j,k}\}_{j\in[n],k\in[M]}$ using outlab
Output y and $\mathsf{st} := \{\mathsf{lab}_{j,k}\}_{j\in[\lambda],k\in[M]}$

Remark 5.7. *For a cleaner presentation, we require the program* Π *to be garbled to satisfy the following properties: (it is already satisfied by the OPRAM compiler we use when composed with full security compiler, but it can be easily satisfied by general programs with preprocessing)*

1. *Read/writes happen simultaneously, that is, at any given time step, either all processors perform a read, or all of them perform a write;*
2. *There are no write collisions, that is, at any given time step of doing wirtes, any pair of processors attempt to write to two different memory locations.*

Correctness. Like the construction in [GOS18], the correctness of the above construction also follows from an inductive argument that for each step state and the database are updated correctly.

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

Adaptive Security. We argue the adaptive security of UGPRAM in Appendix B.

5.6 Full Security

Theorem 5.8. *Assuming the existence of a timed encryption scheme, a puncturable pseudorandom function, an oblivious PRAM scheme with strong localized randomness, and an adaptive garbled PRAM scheme with unprotected memory access with equivocability of size 2, there exists adaptive garbled PRAM, where the time required to garble a database, garble or evaluate a PRAM program, and garble an input grows linearly (up to polylogarithmic factors) with the size of the database, parallel running time of the PRAM program and length of the input respectively.*

To achieve full adaptive security, techniques used to construct adaptive garbled RAM [GOS18] can be easily generalized here, namely, they use ORAM to hide memory access pattern, and timed encryption to be able to encrypt memory content and retrieve it at a later time. Here, in place of oblivious RAM, we use the oblivious PRAM with strong localized randomness to achieve fully secure adaptive garbled PRAM. For timed encryption, we assign each garbled step circuit C an *order* $\pi(C)$ as the timestamp used by the timed encryption, such that the garbled circuits can be evaluated in order $\pi^{-1}(1), \pi^{-1}(2), ...$ sequentially.

Now, we recall the security proof for adaptive garbled RAM from [GOS18]. First, the simulated database is simply a garbled database of all zeroes; the simulated program is a garbled stub program; and the simulated input is a garbled zero string and the correct output XOR the random mask. To change the garbled program into the simulator indistinguishably, they change the RAM program from the last step circuit to the first to their corresponding simulated versions using a sequence of carefully chosen hybrids, and finally by the security of the encryption and the fact that the database and the input was not read by the simulated program at all, they can change the database and the input string to zeroes.

This security proof also naturally extends to our construction of adaptive GPRAM. We change the garbled step circuit one by one from real world to ideal world in the reverse order of π.

Equivocability for UGPRAM. On a high level, equivocability says that one can computationally indistinguishably equivocate poly-logarithmically (but a priori bounded) many step circuits. This property follows naturally from the security proof of the adaptive security. We argue this property in Appendix B.3.

Oblivious PRAM with strong localized randomness. In Appendix C, we will show that the BCP OPRAM scheme [BCP16] can be bootstrapped to have strong localized randomness assuming puncturable PRF, using similar techniques from [GOS18]. We note that our definition of strong localized randomness requires that the "intervals" of step circuit are of size at most constant instead of poly-logarithmic so that it would work in the PRAM case.

Acknowledgements

The authors would like to thank Tsung-Hsuan Hung and Yu-Chi Chen for their helpful discussions in the early stage of this research.

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 *Theory of Cryptography Conference*, pages 3–30. Springer, 2016.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[App17] Benny Applebaum. Garbled circuits as randomized encodings of functions: a primer. In *Tutorials on the Foundations of Cryptography*, pages 1–44. Springer, 2017.

[BCP16] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *Theory of Cryptography Conference*, pages 175–204. Springer, 2016.

[Ben89] Charles H Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, 1989.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *International Workshop on Public Key Cryptography*, pages 501–519. Springer, 2014.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[BLSV18] Zvika Brakerski, Alex Lombardi, Gil Segev, and Vinod Vaikuntanathan. Anonymous ibe, leakage resilience and circular security from new assumptions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 535–564. Springer, 2018.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 280–300. Springer, 2013.

[CCC$^+$16] Yu-Chi Chen, Sherman SM Chow, Kai-Min Chung, Russell WF Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 179–190. ACM, 2016.

[CDG$^+$17] Chongwon Cho, Nico Döttling, Sanjam Garg, Divya Gupta, Peihan Miao, and Antigoni Polychroniadou. Laconic oblivious transfer and its applications. In *Annual International Cryptology Conference*, pages 33–65. Springer, 2017.

[DG17] Nico Döttling and Sanjam Garg. Identity-based encryption from the diffie-hellman assumption. In *Annual International Cryptology Conference*, pages 537–569. Springer, 2017.

[DGHM18] Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, and Daniel Masny. New constructions of identity-based and key-dependent message secure encryption schemes. In *IACR International Workshop on Public Key Cryptography*, pages 3–31. Springer, 2018.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the Association for Computing Machinay. Vol*, 33(4):792–807, 1986.

[GGMP16] Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty RAM computation in constant rounds. In *Theory of Cryptography Conference*, pages 491–520. Springer, 2016.

[GHL$^+$14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 405–422. Springer, 2014.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 210–229. IEEE, 2015.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 449–458. ACM, 2015.

[GOS18] Sanjam Garg, Rafail Ostrovsky, and Akshayaram Srinivasan. Adaptive garbled RAM from laconic oblivious transfer. In *Annual International Cryptology Conference*, pages 515–544. Springer, 2018.

[GS18] Sanjam Garg and Akshayaram Srinivasan. Adaptively secure garbling with near optimal online complexity. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 535–565. Springer, 2018.

[HJO$^+$16] Brett Hemenway, Zahra Jafargholi, Rafail Ostrovsky, Alessandra Scafuro, and Daniel Wichs. Adaptively secure garbled circuits from one-way functions. In *Annual Cryptology Conference*, pages 149–178. Springer, 2016.

[JKK$^+$17] Zahra Jafargholi, Chethan Kamath, Karen Klein, Ilan Komargodski, Krzysztof Pietrzak, and Daniel Wichs. Be adaptive, avoid overcommitting. In *Annual International Cryptology Conference*, pages 133–163. Springer, 2017.

[JW16] Zahra Jafargholi and Daniel Wichs. Adaptive security of Yao's garbled circuits. In *Theory of Cryptography Conference*, pages 433–458. Springer, 2016.

[KPTZ13] Aggelos Kiayias, S Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, page 669, 2013.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs? In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–734. Springer, 2013.

[LO17] Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In *Annual International Cryptology Conference*, pages 66–92. Springer, 2017.

[PTC76] Wolfgang J Paul, Robert Endre Tarjan, and James R Celoni. Space bounds for a game on graphs. *Mathematical systems theory*, 10(1):239–251, 1976.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 475–484. ACM, 2014.

[Yao82] Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

A Security of Adaptive Garbled Circuits

In this section, we prove that the construction of adaptive garbled circuits presented previously satisfies adaptive security.

A.1 Simulators

The proof of security proceeds via a hybrid argument over different circuit configurations which we will first describe in this section. We denote a circuit configuration with $\mathsf{conf} : \left\{\widetilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}, \widetilde{\mathsf{C}}_{i,j}^{\mathsf{hash}}, \widetilde{\mathsf{C}}_{i}^{\mathsf{write}}\right\} \mapsto \{\mathsf{White}, \mathsf{Gray}, \mathsf{Black}\}$[5], which describes the mode of operation of the garbled circuit. White mode corresponds to real garbling, Gray mode corresponds to Input Dependent Simulation (where the step circuit for this gate is in simulation but depends on the input, and is garbled and sent in the e online phase), and Black mode corresponds to Input Independent Simulation which matches the ideal simulator execution.

Valid configurations. We associate the configurations with a directed graph illustrated in Figure 9. We say that a configuration conf is valid if and only if: for every C if $\mathsf{conf}(C) = \mathsf{Black}$ then for every other node C' reachable from C, $\mathsf{conf}(C') = \mathsf{Black}$.

Simulation in a valid configuration. We will now describe the simulator for the adaptive garbling scheme. Note that the final ideal world simulation does not use the circuit C nor the input x. Let I denote the set of circuits that are operating in Gray mode, i.e. $I := \{C : \mathsf{conf}(C) = \mathsf{Gray}\}$.

```
• $\mathsf{SimC}^{\mathsf{Partition}}(1^\lambda, C)$:
    $\mathsf{crs} \leftarrow \ell OT.\mathsf{crsGen}(1^\lambda)$
    $K \leftarrow \mathsf{PRFKeyGen}(1^\lambda)$
    $\{P_i\}_{i\in[\ell]} \leftarrow \mathsf{Partition}(C)$
    Sample $r \leftarrow \{0,1\}^{M\ell}$
    For $i = 1$ to $\ell$ do:
      Let $C_{g,1}, C_{g,2}$ denote the two input gates of gate $g$
      If $\mathsf{conf}(\widetilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}) = \mathsf{White}$ then
        $\widetilde{\mathsf{C}}_{i,k}^{\mathsf{eval}} \leftarrow \mathsf{GCircuit}(1^\lambda, \mathsf{C}_{\mathsf{eval}}^{\mathsf{real}}[\mathsf{crs}, C_{P_{i,k},1}, C_{P_{i,k},2}, P_{i,k}, (r_{C_{P_{i,k},1}}, r_{C_{P_{i,k},2}}, r_{P_{i,k}}),$
                                  $\mathsf{PRF}_K(1,i,k,0), \mathsf{PRF}_K(1,i,k,1)],$
                          $\{\mathsf{PRF}_K(0,i,j,b)\}_{j\in[\lambda],b\in\{0,1\}})$
      If $\mathsf{conf}(\widetilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}) = \mathsf{Black}$ then
        $\widetilde{\mathsf{C}}_{i,k}^{\mathsf{eval}} \leftarrow \mathsf{GCircuit}(1^\lambda, \mathsf{C}_{\mathsf{eval}}^{\mathsf{ideal}}[\mathsf{crs}, C_{P_{i,k},1}, C_{P_{i,k},2}, P_{i,k}, (0, 0, r_{P_{i,k}}),$
                                  $\mathsf{PRF}_K(1,i,k,0), \mathsf{PRF}_K(1,i,k,1)],$
                          $\{\mathsf{PRF}_K(0,i,j,b)\}_{j\in[\lambda],b\in\{0,1\}})$
```

[5]The mapping described here is "symbolic": even though the circuits would be garbled in different ways under different modes, here we map the symbol of the garbled circuits to configurations instead of the garbled circuit itself.

```
      Let $\mathsf{keyEval} = \{\mathsf{PRF}_K(1, i, k, b)\}_{k\in[M], b\in\{0,1\}}$
      Let $\mathsf{keyHash} = \{\mathsf{PRF}_K(2, i, j, b)\}_{j\in[\lambda], b\in\{0,1\}}$
      $\{\widetilde{\mathsf{C}}^{\mathsf{hash}}_{i,j}\}_{j\in[M-1]} \leftarrow \mathsf{GHash}(1^\lambda, \ell OT_{\mathsf{const}}.\mathsf{Hash}, \mathsf{keyEval}, \mathsf{keyHash})$
      If $\mathsf{conf}(\widetilde{\mathsf{C}}^{\mathsf{write}}_{i}) \neq \mathsf{Gray}$ then
        Let $C^{\mathsf{write}}_i = \ell OT.\mathsf{SendWriteBlock}\left(\mathsf{crs}, \cdot, i, \{\mathsf{PRF}_K(0, i+1, j, b)\}_{j\in[\lambda], b\in\{0,1\}}\right)$
        $\widetilde{\mathsf{C}}^{\mathsf{write}}_{i} \leftarrow \mathsf{GCircuit}\left(1^\lambda, C^{\mathsf{write}}_i, \{\mathsf{PRF}_K(2, i, j, b)\}_{j\in[\lambda], b\in\{0,1\}}\right)$
    $(\mathsf{st}_1, c) \leftarrow \mathsf{SEE.SimEnc}\left(I, \left\{\widetilde{\mathsf{C}}^{\mathsf{eval}}_{i,k}, \widetilde{\mathsf{C}}^{\mathsf{hash}}_{i,j}, \widetilde{\mathsf{C}}^{\mathsf{write}}_{i} : \mathsf{conf}(C) \neq \mathsf{Gray}\right\}\right)$
    Output $\widetilde{C} := (\mathsf{crs}, c)$ and $\mathsf{st} := (C, \mathsf{crs}, r, \mathsf{st}_1, \{P_i\}_{i\in[\ell]}, K)$
```

- $\mathsf{SimIn}(\mathsf{st}, x, y)$:

```
    Parse $\mathsf{st} := (C, \mathsf{crs}, r, \mathsf{st}_1, \{P_i\}_{i\in[\ell]}, K)$
    Evaluate the circuit $C(x)$, let $e_j$ be the bit assigned to wire $j$
    For $i \in [\ell]$, let $D_i$ be such that

    $$D_{i,j} = \begin{cases} x \oplus r_{1..n} || 0^{M-n}, & j = 0; \\ E_j \oplus r_{j||0^{\log M}..j||1^{\log M}}, & 0 < j < i; \\ 0, & \text{otherwise}, \end{cases}$$

  where $E_j$ is the bits assigned to wires in $P_j$ of the circuit $C$ computed on input $x$
    Let $\mathsf{d}_i$ be the digest of $D_i$, i.e. $(\mathsf{d}_i, \cdot) := \ell OT.\mathsf{Hash}(\mathsf{crs}, D_g)$.
    For each $\widetilde{\mathsf{C}}^{\mathsf{eval}}_{i,k} \in I$:
      $w \leftarrow e_{P_{i,k}} \oplus r_{P_{i,k}}$
      $\mathsf{out} \leftarrow \ell\mathsf{OTSim}(\mathsf{crs}, D_i, C_{P_{i,k},1}, \ell\mathsf{OTSim}(\mathsf{crs}, D_i, C_{P_{i,k},2}, (w, \mathsf{PRF}_K(1, i, k, w))))$
      $\widetilde{\mathsf{C}}^{\mathsf{eval}}_{i,k} \leftarrow \mathsf{GCircSim}(1^\lambda, 1^{|C_{\mathsf{eval}}|}, \{\mathsf{PRF}_K(0, i, j, d_{i,j})\}_{j\in[\lambda]}, \mathsf{out})$
    For each $\widetilde{\mathsf{C}}^{\mathsf{hash}}_{i,j} \in I$:
      Generate appropriate labels $\mathsf{labs}$ and output $\mathsf{out}$ using $\mathsf{PRF}_K$ for the circuit according to the evaluation
      $\widetilde{\mathsf{C}}^{\mathsf{hash}}_{i,j} \leftarrow \mathsf{GCircSim}(1^\lambda, 1^{|\ell OT_{\mathsf{const}}.\mathsf{Hash}|}, \mathsf{labs}, \mathsf{out})$
    For each $\widetilde{\mathsf{C}}^{\mathsf{write}}_{i} \in I$:
      $\mathsf{out} \leftarrow \ell\mathsf{OTSimWriteBlock}(\mathsf{crs}, D_i, \mathsf{d}_i, i, E_i \oplus r_{i||0^{\log M}..i||1^{\log M}}, \{\mathsf{PRF}_K(0, i+1, j, d_{i+1,j})\}_{j\in[\lambda]})$
      $\widetilde{\mathsf{C}}^{\mathsf{write}}_{i} \leftarrow \mathsf{GCircSim}(1^\lambda, 1^{|\ell OT.\mathsf{SendWriteBlock}|}, \{\mathsf{PRF}_K(2, i, j, d_{i+1,j})\}_{j\in[\lambda]}, \mathsf{out})$
    $\mathsf{key} \leftarrow \mathsf{SEE.SimKey}(\mathsf{st}_1, I)$
    For each $g \in [N - m + 1, N]$:
      Let $\widetilde{\mathsf{C}}^{\mathsf{eval}}_{i,k}$ be the evaluation circuit corresponding to $g$
      If $\mathsf{conf}(\widetilde{\mathsf{C}}^{\mathsf{eval}}_{i,k}) = \mathsf{Black}$ then set $r_g \leftarrow r_g \oplus y_{g-N+m}$
    Output $(\{\mathsf{PRF}_K(0, 1, j, \mathsf{d}_j)\}_{j\in[\lambda]}, r_1 \oplus x_1 || ... || r_n \oplus x_n, \mathsf{key}, r_{N-m+1} || ... || r_N)$
```

A.2 Pebbling Game

Like previous works in constructing adaptive garbling schemes in which garbled circuits talk to each other [HJO+16, GS18, GOS18], our proof of security is also associated with a pebbling game. A pebbling game is a game played on a DAG $G = (V, E)$, where each vertex represents a garbled circuit, and each edge (a, b) represents the garbled circuit a "talks" to garbled circuit b, either directly (a's output is simply b's label) or indirectly (a outputs an ℓOT ciphertext, when received outputs labels for b). We put a gray pebble on a node to represent that the underlying circuit is operating in Gray mode, and a black pebble to represent that the underlying circuit is in Black mode. There are 2 rules for putting/removing the pebbles.

Rule A: We can put a gray pebble on a vertex, if either all of its predecessors are pebbled gray or it's the source code. Looking ahead, since indistinguishability proofs work both ways, we can also remove a gray pebble under the same conditions. We refer to those invocations of the rule as *inverse rule A*.

Rule B: We can put a black pebble on a vertex, if all nodes reachable from the circuit are also pebbled black and all its direct predecessor are pebbled gray.

The goal of the game is to put a black pebble over every single vertex of the graph, which indicates that we have reached the simulation mode. The optimization goal of the pebbling game is to minimize the *pebbling complexity*, which is the maximum number of gray pebbles that are present on the graph over the course of the game. Intuitively, we can think of gray pebbles as a stronger simulated garbled circuit, but also more expensive to use.

A.3 Pebbling Merkle Tree

In this section, we first look at the pebbling complexity of our garbled Merkle tree algorithm given in Section 4.3. Recall that the hierarchy of the garbled Merkle tree is a complete binary tree, with edges from children to their parent.

Lemma A.1. *For any $d \in \mathbf{N}$, there exists a strategy for putting a gray pebble at only the root of the Merkle tree graph of depth d using at most $2d - 1$ gray pebbles and making $\mathsf{poly}(2^d)$ moves.*

Proof. We prove this by induction. For $d = 1$, the root is also the leaf node, therefore we can directly put a gray pebble at the root and we are done.

Assuming the statement is true for $d - 1$, in order to put a gray pebble at the root, we first recursively use the strategy to put a gray pebble at its two children recursively, then put a gray pebble at the root, and finally recursively use the strategy again to remove the two pebbles at its two children. Note that by induction at any point in this process, there can only be at most 2 gray pebbles in each layer of the tree, therefore $2d - 1$ gray pebbles suffice and it requires $O(4^d)$ moves. □

Theorem A.2. *For any $d \in \mathbf{N}$, there exists a strategy for pebbling the Merkle tree graph of depth d according to rules using at most $2d$ pebbles and making $\mathsf{poly}(2^d)$ moves, i.e. the garbled Merkle tree has pebbling complexity $2d$.*

Proof. The strategy is also similar to pebbling the line graph [GS18].

1. Use the strategy in Lemma A.1 to place a gray pebble in the root;
2. If $N > 1$, use the strategy again to place two gray pebbles in its children as well;
3. Replace the gray pebble in the root with a black pebble;
4. Repeat 2-3 for each children recursively until the tree is covered by black pebbles.

Similar to Lemma A.1, at any point in this process, there can only be at most 2 gray pebbles in each layer of the tree, $2d - 1$ gray pebbles suffice and it also requires $O(4^d)$ moves. $\square$

Finally, we note that the fact that such strategy works only depends on the structure of the DAG, but regardless of what each circuit is computing.

A.4 Pebbling Strategy

We now direct our attention back to proving security for our adaptive garbled circuits. We move through different configurations using the same rules as given in Appendix A.2.[6]. We will later prove that changing according to these 2 rules is computationally indistinguishable in Lemmas A.5 and A.6. By a standard hybrid argument, we prove that our construction indeed satisfies the security requirement of adaptive garbled circuits.

Recall that the pebbling graph for our scheme is a series of tree graphs as seen in Figure 9. We will first show a good strategy for pebbling a single layer of this graph. The idea is to use Lemma A.1 and Theorem A.2 as a black box and pebble the layer with this strategy as a subroutine. Therefore we can pebble each layer with pebbling complexity $O(\log M)$. As the whole graph is a "line" of tree graphs, it would be helpful to recall the lemma for pebbling the line graph:

Lemma A.3 ([GS18], Lemma 5.2)**.** *For any $N \in \mathbf{N}$, there exists a strategy for pebbling the line graph $[N]$ according to rule using at most $\log N$ pebbles and making $O(N^{\log_2 3})$ moves.*

Theorem A.4. *Assuming the existence of laconic oblivious transfer scheme, selectively secure circuit garbling scheme, somewhere equivocal encryption scheme and PRFs, there exists an adaptively secure garbled circuit scheme with online complexity $n + m + \mathsf{poly}(\lambda, \log |C|)$ whose parallel evaluation time is $d \cdot \mathsf{poly}(\lambda, \log |C|)$, where n and m is the size of the input and output respectively, and d is the depth of the circuit.*

Proof. First, we will describe our pebbling strategy. The idea is to use Lemma A.3 with some modifications:

1. Every time we need to put/remove a gray pebble on a tree, we invoke Lemma A.1 to put/remove the gray pebble at the end of the tree graph. Note that in the tree graph, only the last node can be the predecessor of any node in the next tree graph, and since we only care about whether the direct predecessor of any given node is gray, it would be sufficient.

[6]Note that in our construction, hash circuits and write circuits are exactly the same under White and Black mode, therefore we can have an additional rule that will help us save a few steps in the hybrids, but as this change does not affect the number of hybrids asymptotically, we will keep things simple and only use these two rules.

2. Every time we need to replace the gray pebble on a tree with black pebbles, we invoke Theorem A.2 to replace the entire tree with black pebbles.

It's not hard to see that we can implement this strategy with the only two rules we have. We will be able to pebble the entire graph using at most $\log \ell + 2 \log M + O(1)$ pebbles (this corresponds to the equivocation parameter for somewhere equivocal encryption) and making $\mathsf{poly}(|C|)$ moves.

Using this strategy yields a sequence of configurations $\mathsf{conf}_0, ..., \mathsf{conf}_\alpha$ for $\alpha = \mathsf{poly}(|C|)$. Let $\mathsf{Hybrid}_{\mathsf{conf}}$ denote the distribution of the output of the simulators using configuration conf. By Lemmas A.5 and A.6, we have that $\mathsf{Hybrid}_{\mathsf{conf}_{i-1}} \overset{c}{\approx} \mathsf{Hybrid}_{\mathsf{conf}_i}$. Finally, note that the initial configuration and the honest evaluation is the same as guaranteed by the simulation with no holes property of somewhere equivocal encryption. This completes the proof of adaptive security. □

A.5 Implementing the Rules

Lemma A.5 (Rule A). *Let* conf *and* conf' *be two valid circuit configurations satisfying the constraints of rule A, then assuming the security of somewhere equivocal encryption, garbling scheme for circuits and updatable* ℓOT*, we have that* $\mathsf{Hybrid}_{\mathsf{conf}} \overset{c}{\approx} \mathsf{Hybrid}_{\mathsf{conf}'}$.

Proof. We prove this via a hybrid argument. Let $\tilde{C}$ be the circuit that are generated differently between conf and conf'.

- $\mathsf{Hybrid}_{\mathsf{conf}}$: This is our starting hybrid.

- Hybrid_1: In this hybrid, we move the generation of $\tilde{C}$ to SimIn but is generated honestly. Computational indistinguishability of $\mathsf{Hybrid}_{\mathsf{conf}}$ and Hybrid_1 reduces directly to the security of somewhere equivocal encryption scheme.

- Hybrid_2: By conditions of rule A, all the predecessors of $\tilde{C}$ must be operating in gray mode. Therefore in this hybrid, we can change all PRF invocations for generating the input labels for $\tilde{C}$ to freshly sampled randomness in SimIn. Computational indistinguishability of Hybrid_1 and Hybrid_2 reduces directly to the security of PRF.

- Hybrid_3: In this hybrid, we will only sample the input labels that will be used and instead use $\mathsf{GCircSim}$ to generate the garbled circuit $\tilde{C}$. Computational indistinguishability of Hybrid_2 and Hybrid_3 reduces directly to the selective security of the garbling scheme.

- Hybrid_4: (for $\tilde{\mathsf{C}}^{\mathsf{eval}}_{i,k}$ only) In this hybrid, we change the output value hardwired in $\tilde{C}$ - instead of using $\ell OT.\mathsf{Send}$, it invokes $\ell\mathsf{OTSim}$. Computational indistinguishability of Hybrid_3 and Hybrid_4 reduces directly to sender privacy of ℓOT.

- Hybrid_5: (for $\tilde{\mathsf{C}}^{\mathsf{eval}}_{i,k}$ only) In this hybrid, we change the output value hardwired in $\ell\mathsf{OTSim}$, i.e. we change the inner $\ell OT.\mathsf{Send}$ to $\ell\mathsf{OTSim}$ as well. Computational indistinguishability of Hybrid_4 and Hybrid_5 also reduces directly to sender privacy of ℓOT.

- Hybrid_6: (for $\widetilde{\mathsf{C}}_i^{\mathsf{write}}$ only) In this hybrid, we change the output value hardwired in $\widetilde{C}$ - instead of using $\ell OT.\mathsf{SendWriteBlock}$, it invokes $\ell\mathsf{OTSimWriteBlock}$. Computational indistinguishability of Hybrid_5 and Hybrid_6 reduces directly to sender privacy of ℓOT in the block writing settings.

- Hybrid_7: In this hybrid, we revert the changes in Hybrid_2, namely we use the PRF to generate the keys. Computational indistinguishability of Hybrid_6 and Hybrid_7 reduces directly to the security of PRF.

Finally observe that Hybrid_7 is the same as $\mathsf{Hybrid}_{\mathsf{conf}'}$. □

Lemma A.6 (Rule B). *Let* conf *and* conf' *be two valid circuit configurations satisfying the constraints of rule B, then assuming the security of somewhere equivocal encryption, garbling scheme for circuits and updatable* ℓOT*, we have that* $\mathsf{Hybrid}_{\mathsf{conf}} \overset{c}{\approx} \mathsf{Hybrid}_{\mathsf{conf}'}$.

Proof. We prove this via a hybrid argument. Let $\widetilde{C}$ be the circuit that are generated differently between conf and conf'. In order to keep the proof close to Lemma A.5, we will proceed the hybrid argument in reverse order, i.e. we will change the circuit from Black mode to Gray mode.

- $\mathsf{Hybrid}_{\mathsf{conf}'}$: This is our starting hybrid.

- Hybrid_1: In this hybrid, we move the generation of $\widetilde{C}$ to SimIn but is generated honestly. Computational indistinguishability of $\mathsf{Hybrid}_{\mathsf{conf}}$ and Hybrid_1 reduces directly to the security of somewhere equivocal encryption scheme.

- Hybrid_2: By conditions of rule A, all the predecessors of $\widetilde{C}$ must be operating in gray mode. Therefore in this hybrid, we can change all PRF invocations for generating the input labels for $\widetilde{C}$ to freshly sampled randomness in SimIn. Computational indistinguishability of Hybrid_1 and Hybrid_2 reduces directly to the security of PRF.

- Hybrid_3: In this hybrid, we will only sample the input labels that will be used and instead use $\mathsf{GCircSim}$ to generate the garbled circuit $\widetilde{C}$. Computational indistinguishability of Hybrid_2 and Hybrid_3 reduces directly to the selective security of the garbling scheme.

- Hybrid_4: (for $\widetilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}$ only) In this hybrid, we change the output value hardwired in $\widetilde{C}$ - instead of using $\ell OT.\mathsf{Send}$, it invokes $\ell\mathsf{OTSim}$. Computational indistinguishability of Hybrid_3 and Hybrid_4 reduces directly to sender privacy of ℓOT.

- Hybrid_5: (for $\widetilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}$ only) In this hybrid, we change the output value hardwired in $\ell\mathsf{OTSim}$, i.e. we change the inner $\ell OT.\mathsf{Send}$ to $\ell\mathsf{OTSim}$ as well. Computational indistinguishability of Hybrid_4 and Hybrid_5 also reduces directly to sender privacy of ℓOT.

- Hybrid_6: (for $\widetilde{\mathsf{C}}_i^{\mathsf{write}}$ only) In this hybrid, we change the output value hardwired in $\widetilde{C}$ - instead of using $\ell OT.\mathsf{SendWriteBlock}$, it invokes $\ell\mathsf{OTSimWriteBlock}$. Computational indistinguishability of Hybrid_5 and Hybrid_6 reduces directly to sender privacy of ℓOT in the block writing settings.

- Hybrid_7: In this hybrid, we revert the changes in Hybrid_2, namely we use the PRF to generate the keys. Computational indistinguishability of Hybrid_6 and Hybrid_7 reduces directly to the security of PRF.

Finally if the circuit is $\tilde{\mathsf{C}}^{\mathsf{eval}}_{i,k}$ (otherwise we are done), observe that the only difference between Hybrid_7 and $\mathsf{Hybrid}_{\mathsf{conf}}$ is how $D_{i+1,P_{i,k}}$ is set. However, we argue that the distributions are in fact identical. Let $g := P_{i,k}$ be the gate evaluated by the circuit and w_g be the value of the wire under honest evaluation.

- If $g \leq N - m$, note that since r_g is not anywhere else we have that the distribution r_g and $r_g \oplus w_g$ are both uniform and identical.
- If $g > N - m$, we have that $r_g = w_g \oplus r'_g$ which is identical to the distribution in $\mathsf{Hybrid}_{\mathsf{conf}}$.

This completes the proof of the lemma. □

B Security of UGPRAM

B.1 Security of Parallel Laconic OT

Now, we prove that the parallel updatable laconic oblivious transfer we constructed is secure.

Theorem B.1 (Sender Privacy against Semi-honest Receiver). *Given that ℓOT_{const} has sender privacy and that the garbled circuit scheme* GCircuit *is secure, the parallel updatable laconic oblivious transfer scheme OThas sender privacy.*

Proof. The proof of sender privacy in the read-only setting in [CDG+17] directly applies since we are using the same construction. For the parallel write setting, we first provide the simulator $\ell\mathsf{OTSimPWrite}$.

```
$\ell\mathsf{OTSimPWrite}\left(\mathsf{crs}, D, L_k, \{m_{k,i,j,\hat{\mathsf{D}}_{i,j}}\}_{i\in[d],j\in[2\lambda]}\right)$:
  $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
  Parse $L_k = (b_{k,1}, b_{k,2}, ..., b_{k,d-1}, t_k)$
  Sample $\left\{\mathsf{Labels}^i_k\right\}_{i\in[d-1]}$ u.a.r.
  Let $\widetilde{\mathsf{Labels}}^i_k$ to be $\{m_{k,i,j,\hat{\mathsf{D}}_{i,j}}\}_{j\in[2\lambda]}$
  $\mathsf{leaf}_k \leftarrow (\hat{\mathsf{D}}_{b_{k,1}\dots b_{k,d-1}0}, \hat{\mathsf{D}}_{b_{k,1}\dots b_{k,d-1}1})$
  $\mathsf{e}^{d-1}_k \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{leaf}_k, 0^*)$
  $\tilde{\mathsf{e}}^{d-1}_k \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}\left(\mathsf{crs}, \mathsf{leaf}_k, \widetilde{\mathsf{Labels}}^d_k\right)$
  For $i = d-1$ downto 1:
    $\tilde{\mathsf{C}}^i_k \leftarrow \mathsf{GCircSim}\left(1^\lambda, 1^{|\mathsf{C}^{\mathsf{trav}}|}, \left\{\mathsf{Labels}^i_k\right\}_{i\in[d]}, (\mathsf{e}^i_k, \tilde{\mathsf{e}}^i_k)\right)$
    $\mathsf{sbl}^i_k \leftarrow (\hat{\mathsf{D}}_{b_{k,1}\dots b_{k,i-1}0}, \hat{\mathsf{D}}_{b_{k,1}\dots b_{k,i-1}1})$
    $\mathsf{e}^{i-1}_k \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i_k, \widetilde{\mathsf{Labels}}^i_k)$
```

$\tilde{\mathsf{e}}_k^{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}\left(\mathsf{crs}, \mathsf{sbl}_k^i, \widetilde{\mathsf{Labels}}_k^i\right)$

Output $\mathsf{e}_k = \left(\mathsf{e}_k^0, \tilde{\mathsf{e}}_k^0, \{\tilde{\mathsf{C}}_k^i\}_{i\in[d-1]}\right)$

A very similar indistinguishability proof can also be applied to the parallel write settings, namely, we can prove indistinguishability for the first $2d-1$ hybrids using the argument from [CDG+17]. Finally, note that the $(2d-1)$-th hybrid and is exactly the same as the simulated experiment. □

B.2 Adaptive Security

Adaptive security for unprotected memory access garbled PRAM can be argued the same way as Theorem A.4, except with a few changes. Recall that in order to prove adaptive security for our construction of garbled circuits, we start with the real world construction, and use a sequence of hybrids to change each evaluation circuit from the real evaluation mode to the ideal evaluation mode, so that we erase the real functionality that we are computing inside the garbled circuit. At a high level, we abstract this hybrid sequence into a pebbling game as described in Appendix A.2. In each configuration of the pebbling game, we assign each step circuit a mode, where White corresponds to real garbling, Gray corresponds to equivocated garbling, and Black corresponds to ideal garbling. To finish the security proof, we show that each pebbling rule can be implemented, i.e. the output distributions are computationally indistinguishable before and after the change.

In this section, we will highlight the key differences in the security proof.

The pebbling graph for this garbling scheme actually looks almost identical to that for our garbled circuits. In this construction, the width of the garbled Merkle tree is $O(nM)$, and the length of the line is $O(T)$. Thus using the same pebbling strategy as in Theorem A.4, we can pebble the pebbling graph using at most $\log T + \log n + \log M + O(1)$ pebbles.

Now to complete the security proof, we will show how to implement the rules for the pebbling game for adaptive garbled PRAM with UMA. The simulator SimMemory is identical to Memory. We will layout the simulator (SimProgram, SimInput) through describing the changes we made in the security proof. We define circuit configurations and valid configurations analogous to Appendix A.1, the only difference is that the circuits in the configuration are replaced with the circuits in this construction. Let $\mathsf{Hybrid}_{\mathsf{conf}}$ denote the distribution of the output of the simulators using configuration conf.

At the beginning of the hybrid argument, the simulators are identical to the honest evaluation of Program and Input, except that we replace the invocations of somewhere equivocal encryption with its simulators, analogous to Appendix A.1. The distribution of this hybrid is completely identical to that of the honest evaluation, due to simulation with no holes property of somewhere equivocal encryption.

Lemma B.2 (Rule A). *Let* conf *and* conf$'$ *be two valid circuit configurations satisfying the constraints of rule A, then assuming the security of somewhere equivocal encryption, garbling scheme for circuits and updatable ℓOT, we have that* $\mathsf{Hybrid}_{\mathsf{conf}} \overset{c}{\approx} \mathsf{Hybrid}_{\mathsf{conf}'}$.

Proof. We prove this via a hybrid argument. Recall that for Rule A, we equivocate a circuit so that we can determine what to send while we garble the input. Let $\tilde{C}$ be the circuit that are generated differently between conf and conf$'$.

- $\mathsf{Hybrid}_{\mathsf{conf}}$: This is our starting hybrid.
- Hybrid_1: In this hybrid, we move the generation of $\tilde{C}$ from $\mathsf{SimProgram}$ to $\mathsf{SimInput}$ but is generated honestly. Computational indistinguishability of $\mathsf{Hybrid}_{\mathsf{conf}}$ and Hybrid_1 reduces directly to the security of somewhere equivocal encryption scheme.
- Hybrid_2: By conditions of rule A, all the predecessors of $\tilde{C}$ must be operating in gray mode. Therefore in this hybrid, we can change all PRF invocations for generating the input labels for $\tilde{C}$ to freshly sampled randomness in $\mathsf{SimInput}$. Computational indistinguishability of Hybrid_1 and Hybrid_2 reduces directly to the security of PRF.
- Hybrid_3: In this hybrid, we will only sample the input labels that will be used and instead use $\mathsf{GCircSim}$ to generate the garbled circuit $\tilde{C}$. Computational indistinguishability of Hybrid_2 and Hybrid_3 reduces directly to the selective security of the garbling scheme.
- Hybrid_4: (for $\tilde{\mathsf{C}}^{\mathsf{eval}}_{\tau,k}$ only) In this hybrid, we change the output value hardwired in $\tilde{C}$ - instead of using $\ell OT.\mathsf{Send}$, it invokes $\ell\mathsf{OTSim}$. Computational indistinguishability of Hybrid_3 and Hybrid_4 reduces directly to sender privacy of ℓOT.
- Hybrid_5: (for $\tilde{\mathsf{C}}^{\mathsf{eval}}_{\tau,k}$ only) In this hybrid, we change the output value hardwired in $\tilde{C}$ - instead of using $\ell OT.\mathsf{SendPWrite}$, it invokes $\ell\mathsf{OTSimPWrite}$. Computational indistinguishability of Hybrid_4 and Hybrid_5 reduces directly to sender privacy for parallel writes of ℓOT.
- Hybrid_6 ($\mathsf{Hybrid}_{\mathsf{conf}'}$): In this hybrid, we revert the changes in Hybrid_2, namely we use the PRF to generate the keys. Computational indistinguishability of Hybrid_5 and Hybrid_6 reduces directly to the security of PRF.

This completes the proof of the lemma. $\square$

Lemma B.3 (Rule B)**.** *Let* conf *and* conf' *be two valid circuit configurations satisfying the constraints of rule B, then assuming the security of somewhere equivocal encryption, garbling scheme for circuits and updatable* ℓOT*, we have that* $\mathsf{Hybrid}_{\mathsf{conf}} \overset{c}{\approx} \mathsf{Hybrid}_{\mathsf{conf}'}$.

Proof. We prove this via a hybrid argument. For Rule B, we change a previously equivocated circuit to the corresponding step circuit in the other garbled program. Let $\tilde{C}$ be the circuit that are generated differently between conf and conf'. In order to keep the proof close to Lemma B.2, we will proceed the hybrid argument in reverse order, i.e. we will change the circuit from Black mode to Gray mode.

- $\mathsf{Hybrid}_{\mathsf{conf}'}$: This is our starting hybrid. Note that the step circuit $\tilde{C}$ comes from the second program that we are given.
- Hybrid_1: In this hybrid, we move the generation of $\tilde{C}$ to $\mathsf{SimInput}$ but is generated honestly. Computational indistinguishability of $\mathsf{Hybrid}_{\mathsf{conf}}$ and Hybrid_1 reduces directly to the security of somewhere equivocal encryption scheme.

- Hybrid_2: By conditions of rule A, all the predecessors of $\tilde{C}$ must be operating in gray mode. Therefore in this hybrid, we can change all PRF invocations for generating the input labels for $\tilde{C}$ to freshly sampled randomness in $\mathsf{SimInput}$. Computational indistinguishability of Hybrid_1 and Hybrid_2 reduces directly to the security of PRF.

- Hybrid_3: In this hybrid, we will only sample the input labels that will be used and instead use $\mathsf{GCircSim}$ to generate the garbled circuit $\tilde{C}$. Computational indistinguishability of Hybrid_2 and Hybrid_3 reduces directly to the selective security of the garbling scheme.

- Hybrid_4: (for $\tilde{\mathsf{C}}^{\mathsf{eval}}_{\tau,k}$ only) In this hybrid, we change the output value hardwired in $\tilde{C}$ - instead of using $\ell OT.\mathsf{Send}$, it invokes $\ell\mathsf{OTSim}$. Computational indistinguishability of Hybrid_3 and Hybrid_4 reduces directly to sender privacy of ℓOT.

- Hybrid_5: (for $\tilde{\mathsf{C}}^{\mathsf{eval}}_{\tau,k}$ only) In this hybrid, we change the output value hardwired in $\tilde{C}$ - instead of using $\ell OT.\mathsf{SendPWrite}$, it invokes $\ell\mathsf{OTSimPWrite}$. Computational indistinguishability of Hybrid_4 and Hybrid_5 reduces directly to sender privacy for parallel writes of ℓOT.

- Hybrid_6 ($\mathsf{Hybrid}_{\mathsf{conf}}$): In this hybrid, we revert the changes in Hybrid_2, namely we use the PRF to generate the keys. Computational indistinguishability of Hybrid_5 and Hybrid_6 reduces directly to the security of PRF.

This completes the proof of the lemma. □

B.3 Equivocability of UGPRAM

Similar to [GOS18], our construction of fully secure GPRAM also requires UGPRAM to have the property of equivocability. On a high level, equivocability says that we can equivocate up to Q blocks of messages (garbled circuits) indistinguishably. Looking ahead, in the full security proof, we actually only need to equivocate at most 2 garbled circuits simultaneously, which is always satisfied by our construction.

Definition B.4 (Equivocability of UGPRAM)**.** *We call* UGPRAM *has equivocability of size Q if there exists a stateful simulator* Sim *such that for any non-uniform PPT stateful adversary* $\mathcal{A}$*, s.t.*

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

where $\mathsf{EquivExpt}^{\mathsf{real}}$ *and* $\mathsf{EquivExpt}^{\mathsf{ideal}}$ *are described in fig. 18.*

Proof sketch of equivocability. To prove that our construction has equivocability of size 2, we employ a similar sequence of hybrids as in adaptive security proof, namely, for each circuit that we needs to equivocate, we use Rule A (Lemma B.2) and our pebbling strategy in the pebbling game to put a gray pebble on that circuit. It is not hard to see that our construction will always have equivocability of size 2 as long as there are at least 2 pieces of garbled circuits.

```
$\mathsf{EquivExpt}^{\tau\in\{\text{real, ideal}\}}(1^\lambda, \mathcal{A})$
  $(D, 1^T) \leftarrow \mathcal{A}(1^\lambda)$
  If $\tau$ is real, $(\tilde{D}, SK) \leftarrow \mathsf{Memory}(1^\lambda, D)$
  If $\tau$ is ideal, $\tilde{D} \leftarrow \mathsf{Sim}(1^\lambda, D)$
  For $i \in [T]$ do
    $(\Pi_i, I) \leftarrow \mathcal{A}(\tilde{D}, \{\tilde{\Pi}_j, \tilde{x}_j\}_{j\in[i-1]})$ where $I \subset [|\Pi_i|]$ and $|I| \leq Q$
    If $\tau$ is real, $\tilde{\Pi}_i \leftarrow \mathsf{Program}(SK, i, \Pi_i)$
    If $\tau$ is ideal, $\tilde{\Pi}_i \leftarrow \mathsf{Sim}(i, \Pi_i - I)$
    $x_i \leftarrow \mathcal{A}_2(\mathsf{st}, \{\tilde{\Pi}_j, \tilde{x}_j\}_{j\in[i-1]}, \tilde{\Pi}_i)$
    If $\tau$ is real, $\tilde{x}_i \leftarrow \mathsf{Input}(SK, i, x_i)$
    If $\tau$ is ideal, $\tilde{x}_i \leftarrow \mathsf{Sim}(i, x_i, \Pi_i - I, \{y_C\}_{C\in I})$ where $y_C$ is the output of $C$ when $\Pi_i$
is executed with $x_i$
  Output $\mathcal{A}_3(\mathsf{st}, \mathsf{e})$
```

Figure 18: Equivocability Security Game

C Oblivious Parallel RAM with Strong Localized Randomness

The notion of oblivious parallel RAM (OPRAM) compiler is introduced by [BCP16], which extends the standard ORAM compiler for compiling parallel programs.

Definition C.1 (Oblivious Parallel RAM). *An oblivious Parallel RAM scheme is a pair of PRAM programs* $(\mathsf{OPProg}, \mathsf{OPData})$ *with the following syntax:*

- $\Pi^* \leftarrow \mathsf{OPProg}(1^\lambda, |D|, 1^T, \Pi)$*: Given security parameter* λ*, the size of the memory* $|D|$*, a PRAM program* Π *that runs in parallel time* T *with* M *processors,* OPProg *outputs an probablistic oblivious program* Π^* *which uses also* M *processors and* D^* *as RAM. A probabilistic PRAM program is modeled exactly as a deterministic program except that each step ccircuit additionally take random coins as input;*
- $D^* \leftarrow \mathsf{OPData}(1^\lambda, D)$*: Given security parameter* λ*, the contents of the database* D*, outputs the oblivious database* D^**.*[7]

Correctness. *For any sequence of programs* $\Pi_1, ..., \Pi_\ell$ *and inputs* $x_1, ..., x_\ell$*,*

$$\Pr\left[(\Pi_1(x_1), ..., \Pi_\ell(x_\ell))^D = (\Pi_1^*(x_1^*), ..., \Pi_\ell^*(x_\ell^*))^{D^*}\right] \geq 1 - \mathsf{negl}(\lambda).$$

Obliviousness. *There exists a PPT simulator* OPSim *such that for any sequence of PRAM programs and inputs* $\Pi_1, \Pi_2, ..., \Pi_\ell$ *with* M *processors whose running times are bounded by* $T_1, ..., T_\ell$ *respectively, the memory access patterns produced by sequentially running these programs are computationally indistinguishable to* $\mathsf{OPSim}(1^\lambda, 1^{T_1}, ..., 1^{T_\ell})$*.*

[7]We can always implement OPData using OPProg only, by compiling a PRAM program that writes D to the memory and evaluating the compiled program honestly.

Efficiency. *For any input x, the parallel running time of* OPProg *should be* $T \cdot \mathsf{poly}(\lambda, \log M, \log |D|)$*, and the parallel running time of* OPData *should be* $|D|/M \cdot \mathsf{poly}(\lambda, \log M, \log |D|)$*. Finally, the parallel running time* T^* *of* $\Pi^{*D^*}(x)$ *is bounded by* $T \cdot \mathsf{poly}(\lambda, \log M, \log |D|)$*, where* T *is the parallel running time of* $\Pi^D(x)$*.*

Remark C.2. *In order to simplify the presentation, we implicitly operate over large blocks, as required by OPRAM. Our construction does not care about block sizes and our efficiency will hold as long as the block sizes do not exceed* $\mathsf{poly}(\lambda, \log M, \log |D|)$*, which is satisfied by [BCP16].*

In our construction, we also need the OPRAM compiler to have the property of being collision free.

Definition C.3 (Collision-Free)**.** *An OPRAM compiler is said to be collision free if for any program* Π*, the program that the OPRAM compiler outputs* $\Pi^\star$ *has the property that if any two processors access the same data address in the same timestep, both of their operations are reads.*

Remark C.4. *As noted in [BCP16], it is possible to construct an OPRAM compiler with neither read collisions nor write collisions. For our construction, it is sufficient to have write-collision-freeness, but by plugging the stronger OPRAM compiler into our construction of GPRAM, our GPRAM can also satisfy this stronger notion of collision-freeness.*

Theorem C.5. *There exists a collision-free OPRAM compiler.*

Proof. [BCP16] gave a construction of collision-free OPRAM compiler. We note that our definition of OPRAM is different from the definition there in the following ways:

1. We assume all CPUs are active at all time by making idle CPUs spin waiting, therefore, we do not need activation patterns in the definition.

2. Our correctness requires the protocol to succeed with probability $1 - \mathsf{negl}(\lambda)$ instead of $1 - \mathsf{negl}(|D|)$. This can be achieved by running $O(\lambda)$ parallel copies of OPRAM of the original compiler.

3. Our definition allows for persistent database, namely, we need to evaluate multiple programs. We note that the construction from [BCP16] already can satisfy this generalization.

4. Our obliviousness is simulation-based instead of indistinguishability-based. The construction from [BCP16] can satisfy this generalization, since we can implement the simulator by compiling a stub program that does reads and writes and invoke the indistinguishability security argument.

Finally, we note that this OPRAM compiler is statistically secure, meaning that the output of the simulator is actually statistically indistinguishable to the real memory access pattern. □

For our construction of adaptively secure garbled PRAM, we need an additional property called as strong localized randomness property, similar to the stronger formalization from [GOS18].

Definition C.6 (Strong Localized Randomness)**.** *An OPRAM compiler is said to have strong localized randomness property if given program* Π *(given as a set of all CPU step circuits and their layout) and input* x*,*

1. *There exists an efficiently computable (from* C, Π *but not* x*) function* $I_\Pi(C)$*, given as input any step circuit* $C \in \Pi^*$*, outputs an interval of size at most* $\mathsf{poly}(\log M, \log |D|, \lambda)$*, s.t. the random tape accessed by* C *is given by* $R_{I_\Pi(C)}$*, meaning the random tape restricted to the interval* I_j*;*

2. *For every* $C, C' \in \Pi^*$ *and* $C \neq C'$*,* $I_\Pi(C) \cap I_\Pi(C') = \emptyset$*.*

3. *For every* $C \in \Pi^*$*, there exists a (possibly empty) set* $P_\Pi(C)$ *of size at most* $O(1)$*, s.t. given* $R_{\neg I_\Pi(P_\Pi(C))}$ *(where* $R_{\neg I_\Pi(P_\Pi(C))}$ *denotes the content of the random tape except in positions* $\bigcup_{C' \in P_\Pi(C)} I_\Pi(C')$*) and the outputs of step circuits in* $P_\Pi(C)$*, the memory access pattern of* C *is computationally indistinguishable to random.*

C.1 Puncturable PRF

We recall the definition of puncturable PRF from [GOS18].

Definition C.7. *A puncturable pseudorandom function (PRF)* PP *is a tuple of PPT algorithms* $(\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Punc})$ *with the following properties:*

- ***Functionality is preserved under puncturing****: For all* λ, x, y *such that* $x \neq y$*,*

$$\Pr[\mathsf{Eval}(K[x], y) = \mathsf{Eval}(K, y))] = 1,$$

where $K \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and* $K[x] \leftarrow \mathsf{Punc}(K, x)$*.*

- ***Pseudorandomness at punctured points****: For all* λ, x *and for all polynomial sized adversaries* $\mathcal{A}$*,*

$$|\Pr[\mathcal{A}(K[x], \mathsf{Eval}(K, x)) = 1] - \Pr[\mathcal{A}(K[x], r) = 1]| \leq \mathsf{negl}(\lambda),$$

where $K \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and* $K[x] \leftarrow \mathsf{Punc}(K, x)$ *and* r *is a random string of length* $|\mathsf{Eval}(K, x)|$*.*

Theorem C.8 ([GGM86, BW13, BGI14, KPTZ13, SW14])**.** *Assuming the existence of one-way functions, there exists puncturable pseudorandom functions.*

C.2 Strong Localized Randomness

In this section, we show that OPRAM construction from [BCP16], instantiated with the following special encryption scheme, constructed from puncturable PRF PP, satisfies the strong localized randomness property.

1. $\mathsf{KeyGen}(1^\lambda)$: Sample $K \leftarrow \mathsf{PP.KeyGen}(1^\lambda)$;

2. $\mathsf{Enc}(K, \tau, v)$: The encryption algorithm takes as input the key K, the timestep τ and the value v. It samples a random string $r \leftarrow \{0,1\}^\lambda$ and outputs $(\tau || r, \mathsf{PP.Eval}(K, (\tau || r)) \oplus v)$ as the ciphertext.

3. $\mathsf{Dec}(K, c)$: The ciphertext c is parsed as (c_1, c_2) and the value v is recovered by computing $\mathsf{PP.Eval}(K, c_1) \oplus c_2$.

Now, we recall the BCP OPRAM scheme [BCP16]. In their compiler, the database D^* is maintained as a complete binary tree of depth $\log(|D|/\alpha)$. The memory cell associated with leaf ℓ will be stored at one of the nodes on the path from the root to the leaf ℓ. Each parallel memory access operation $\mathsf{Access}(r, v)$ at time step τ is replaced by a subrouting $\mathsf{OPAccess}(r, v)$, where processor $i \in [M]$ requests a data cell r_i (within some α-block $b_i \in [|D|/\alpha]$) and some action to be taken (either τ to denote read, or v_i to denote rewriting cell r_i with value v_i). The subroutine $\mathsf{OPAccess}$ does the following:

1. **Conflict Resolution:** Run $\mathsf{OblivAgg}$ on inputs $\{(b_i, v_i)\}_{i\in[m]}$ to select a unique representative $\mathsf{rep}(b_i)$ for each queried block b_i and aggregate all CPU instructions for this b_i (denoted $\bar{v}_i$).

2. **(Recursive) Access to Position Map:** Each representative CPU $\mathsf{rep}(b_i)$ samples a fresh random leaf id $\ell'_i \leftarrow [|D|/\alpha]$ in the tree and performs a (recursive) Read/Write access command on the position map database to fetch the current position map value ℓ_i for block b_i and rewrite it with the newly sampled value ℓ'_i. Each dummy CPU performs an arbitrary dummy access, e.g. reading block 1 and rewrite it with some garbage.

3. **Look Up Current Memory Values:** Each CPU $\mathsf{rep}(b_i)$ fetches memory from the database nodes down the path to leaf ℓ_i; when b_i is found, it copies its value v_i into local memory. Each dummy CPU chooses a random path and amke analogous dummy data fetches along it, ignoring all read values.

4. **Remove Old Data:** Aggregate instructions across CPUs accessing the same "buckets" of memory (corresponding to nodes of the tree) on the server side. For each bucket b to be modified, the CPU with the *smallest* id from those who wish to modify b executes the aggregated block-removal instruction. Note that this aggregation step is purely for correctness and not security.

5. **Insert Updated Data into Database in *Parallel*:** Run Route on inputs $\{(m, (\mathsf{msg}_i, \mathsf{addr}_i))\}_{i\in[M]}$, where for each $\mathsf{rep}(b_i)$, $\mathsf{msg}_i = (b_i, \bar{v}_i, \ell'_i)$ (i.e., updated block data) and $\mathsf{addr}_i = [\ell'_i]_{\log M}$ (i.e., level-($\log M$)-truncation of the path ℓ'_i), and for each dummy CPU, $\mathsf{msg}_i = \mathsf{addr}_i = \emptyset$.

6. **Flush the ORAM Database:** In parallel, each CPU initiates an independent flush of the ORAM tree. (Recall that this corresponds to selecting a random path down the tree, and pushing all data blocks in this path as far as they will go). To implement the simultaneous flush commands, as before, commands are aggregated across CPUs for each bucket to be modified, and the CPU with the smallest id performs the corresponding aggregated set of commands. (For example, all CPUs will wish to access to root node in their flush; the aggregation of all corresponding commands to the root node data will be executed by the lowest-numbered CPU who wishes to access this bucket, in this case CPU 1).

7. **Return Output:** Run OblivMCast on inputs $\{(b_i, \hat{v}_i)\}_{i\in[m]}$ (where for dummy CPUs, $b_i = \hat{v}_i = \emptyset$) to communicate the *original* (pre-updated) value of each data block b_i to the subset of CPUs that originally requested it.

For the full description of OblivAgg, Route, and OblivMCast, see [BCP16]. Here, we simply make the observation that all 3 parallel algorithms are deterministic and therefore irrelevant to our discussions below.

Remark C.9 (Local CPU Communcations)**.** *In their construction of OPRAM, they also use local CPU communications that are deterministic and independent of input (i.e. obliviousness). This can be emulated in the standard PRAM model with constant overhead by communicating through memory, or composed directly into our construction (its obliviousness guarantees that it would not break the security). For the rest of the discussion, we will work with the first case (memory-emulated communications) as it is more general.*

Modified OPRAM scheme. The modification that we make to BCP OPRAM scheme is that every values $\{v_i\}_{i\in[M]}$ that is being written to the database at timestep τ is encrypted using the special encryption scheme with respect to the timestep τ. The encryption key is sampled outside of the scheme. Note that these values also include all the temporary memory used for emulating local CPU communications.

Theorem C.10. *Assuming the existence of a puncturable pseudorandom function, there exists a collision-free oblivious PRAM scheme with strong localized randomness.*

Proof. The original OPRAM scheme already satisfies collision-freeness. Since the only change we made is composing it with an encryption layer at the database, correctness and efficiency follows from the correctness and efficiency of both the original scheme and those of our special encryption scheme, and obliviousness preserves as there is no change to its read/write access pattern.

We will argue that our modification satisfies strong localized randomness. Every step samples fresh independent randomness, which directly shows the first two properties in Definition C.6.

We now argue the third property. Note that the memory locations accessed by oblivious inter-CPU communications are determinstic and independent of input, and those accessed by flushing are freshly independently sampled. The only issue is step 3, where the path used by the lookup is read from the position map. In order to make them random and independent, we need to use the special encryption scheme to change the position map. For a particular memory access to a location L, let C be the step circuit (if there is any) such that the value of the position map for location L is last sampled by C. Now, instead of pushing the correct leaf node in the position map it samples, we use the security of the special encryption scheme to push a junk value. With this change, the memory locations accessed by step 3 are also random. We can make the computational indistinguishability argument even when using the encryption key punctured at where the encryption is invoked. $\square$

D Fully Secure GPRAM

D.1 Range Constrained PRF

We use the definition of Range Constrained PRF from [GOS18].

Definition D.1. *A Range Constrained PRF* RC *is a tuple of PPT algorithms* $(\mathsf{KeyGen}, \mathsf{Constrain}, \mathsf{Eval})$ *with the following syntax:*

- $K \leftarrow \mathsf{KeyGen}(1^\lambda)$*: It takes the security parameter* λ *as input and generates a PRF key* K*. We implicitly assume that the key* K *defines an efficiently computable function* $\mathsf{PRF}_K : \{0,1\}^{n(\lambda)} \mapsto \{0,1\}^\lambda$*. For brevity, we use* n *to denote* $n(\lambda)$*.*
- $K[T] \leftarrow \mathsf{Constrain}(K, T)$*: It takes the PRF key* K *and a value* $T \in \{0,1\}^n$*, and deterministically outputs a constrained key* $K[T]$*.*
- $y \leftarrow \mathsf{Eval}(K[T], x)$*: It takes as input a constrained key* $K[T]$ *and an input* x *and outputs either a string* y *or* $\perp$*.*

Correctness. *We say that a range constrained PRF to be correct if for all* $K, T \in \{0,1\}^n$ *and* $x \in [0, T]$*, we have* $\mathsf{Eval}(K[T], x) = \mathsf{PRF}_K(x)$*, where* $K[T] \leftarrow \mathsf{Constrain}(K, T)$*.*

Security. *For security, we require that for any* $T_1, ..., T_\ell \in \{0,1\}^n$ *and any* $y > T_i$ *for all* $i \in [\ell]$*,*

$$\{\{K[T_i]\}_{i \in [\ell]}, \mathsf{PRF}_K(y)\} \overset{c}{\approx} \{\{K[T_i]\}_{i \in [\ell]}, r\},$$

where $K \leftarrow \mathsf{KeyGen}(1^\lambda), K[T_i] \leftarrow \mathsf{Constrain}(K, T_i)$ *for all* $i \in [\ell]$ *and* r *is a random bit string of length* λ*.*

Theorem D.2 ([GOS18])**.** *Assuming the existence of one-way functions, there exists a construction of range-constrained PRFs.*

D.2 Timed Encryption

Timed encryption schemes are introduced by [GOS18] for constructing adaptive garbled RAM. At a high level, it is a symmetric key encryption scheme where every message is encrypted with respect to a timestamp time. Additionally, there is a special algorithm that time-constrains the encryption key K by time time'. This time constrained key $K[\mathsf{time}']$ can then be used to decrypt any ciphertext that is encrypted with respect to timestamp $\mathsf{time} < \mathsf{time}'$, but encryptions "in the future" retains their semantic security.

Definition D.3. *A timed encryption scheme* TE *is a tuple of polynomial-time algorithms* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Constrain}, \mathsf{StrongConstrain})$ *with the following syntax:*

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

- $K^S[\mathsf{time}] \leftarrow \mathsf{StrongConstrain}(K, \mathsf{time})$*: It is a deterministic algorithm that takes as input a key* K *and a timestamp* $\mathsf{time} \in [0, 2^\lambda)$ *and outputs a strongly time-constrained key* $K^S[\mathsf{time}]$*;*

- $c \leftarrow \mathsf{Enc}(K, \mathsf{time}, m)$*: It is a randomized algorithm that takes a key* K*, a timestamp* time *and a message* m *as input, and outputs either the corresponding ciphertext* c *or* $\perp$*;*

- $m \leftarrow \mathsf{Dec}(K, c)$*: It is a deterministic algorithm that takes a key* K *and a ciphertext* c *as input and outputs either the corresponding message* m *or* $\perp$*.*

Correctness. *For any message* m *and timestamps* $\mathsf{time}_1 \leq \mathsf{time}_2$*:*

$$\Pr[\mathsf{Dec}(K[\mathsf{time}_2], \mathsf{Enc}(K, \mathsf{time}_1, m)) = m] = 1,$$

where $K \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and* $K[\mathsf{time}_2] \leftarrow \mathsf{Constrain}(K, \mathsf{time}_2)$*.*

Encrypting with Constrained Key. *For any message* m *and timestamps* $\mathsf{time}_1 \leq \mathsf{time}_2$*, the distributions* $\{\mathsf{Enc}(K, \mathsf{time}_1, m)\}$ *and* $\{\mathsf{Enc}(K[\mathsf{time}_2], \mathsf{time}_1, m)\}$ *are identical, where* $K \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and* $K[\mathsf{time}_2] \leftarrow \mathsf{Constrain}(K, \mathsf{time}_2)$*.*

Encrypting with Strongly Constrained Key. *For any message* m *and timestamps* $\mathsf{time}_1 \leq \mathsf{time}_2$*, the distributions* $\{\mathsf{Enc}(K, \mathsf{time}, m)\}$ *and* $\{\mathsf{Enc}(K^S[\mathsf{time}], \mathsf{time}, m)\}$ *are identical, where* $K \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and* $K[\mathsf{time}] \leftarrow \mathsf{StrongConstrain}(K, \mathsf{time})$*.*

Security. *For any two messages* m_0, m_1 *and timestamps* $(\mathsf{time}, \{\mathsf{time}_i, \mathsf{time}'_i\}_{i \in [t]})$ *such that* $\mathsf{time}_i < \mathsf{time}$ *and* $\mathsf{time}'_i \neq \mathsf{time}$ *for all* $i \in [t]$*:*

$$\{\mathsf{keys}, \mathsf{Enc}(K, \mathsf{time}, m_0)\} \overset{c}{\approx} \{\mathsf{keys}, \mathsf{Enc}(K, \mathsf{time}, m_1)\},$$

where $K \leftarrow \mathsf{KeyGen}(1^\lambda)$*,* $\mathsf{keys} = \{K[\mathsf{time}_i], K^S[\mathsf{time}'_i]\}_{i \in [t]}$*, and* $K[\mathsf{time}_i] \leftarrow \mathsf{Constrain}(K, \mathsf{time}_i), K^S[\mathsf{time}'_i] \leftarrow \mathsf{StrongConstrain}(K, \mathsf{time}'_i)$ *for* $i \in [t]$*.*

Theorem D.4. *Assuming the existence of one-way functions, there exists a construction of timed encryption.*

Proof. Let $(\mathsf{SK.Enc}, \mathsf{SK.Dec})$ be a symmetric key encryption scheme that uses a λ-bit random string as its key. Let RC be a range-constrained PRF. We describe the construction below.

- $\mathsf{KeyGen}(1^\lambda)$: Sample $K \leftarrow \mathsf{RC.KeyGen}(1^\lambda)$ defining $\mathsf{PRF}_K : \{0,1\}^\lambda \mapsto \{0,1\}^\lambda$ and output K.

- $\mathsf{Constrain}(K, \mathsf{time})$: Output $\mathsf{RC.Constrain}(K, \mathsf{time})$.

- $\mathsf{StrongConstrain}(K, \mathsf{time})$: Output $\mathsf{PRF}_K(\mathsf{time})$.

- $\mathsf{Enc}(K, \mathsf{time}, m)$: Let $SK \leftarrow K$ if K is strongly constrained, otherwise compute $SK \leftarrow \mathsf{PRF}_K(\mathsf{time})$. output $(\mathsf{time}, \mathsf{SK.Enc}(SK, m))$.

```
**Step Circuit** $\mathsf{SC}^{\tau}$
  **Hardwired Values/Circuit**: $i, \tau, K[i,\tau,1], I, K', r', C$
  **Input**: $c_{\mathsf{CPU}}, x$
  $\mathsf{rData} \leftarrow \mathsf{TE.Dec}(K[i,\tau,1], x)$
  $\mathsf{state} \leftarrow \mathsf{TE.Dec}(K[i,\tau,1], c_{\mathsf{CPU}})$
  $R_I \leftarrow \mathsf{PP.Eval}(K', I)$
  $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) \leftarrow C(\mathsf{state}, \mathsf{rData}; R_I)$
  If $C$ is an output circuit, $c'_{\mathsf{CPU}} \leftarrow \mathsf{state}' \oplus \mathsf{r}'$
  Else, $c'_{\mathsf{CPU}} \leftarrow \mathsf{TE.Enc}(K[i,\tau,1], (i,\tau,0), \mathsf{state}')$
  If $\tau$ is at the end of $\ell OT.\mathsf{UpdatePWrite}$, $\mathsf{d} \leftarrow \mathsf{tr}_k$
  If $\mathsf{R/W} = \mathsf{write}$
    $x' \leftarrow \mathsf{TE.Enc}(K[i,\tau,1], (i,\tau,1), \mathsf{wData})$
    Output $(c'_{\mathsf{CPU}}, \mathsf{R/W}, L, x')$
  Else output $(c'_{\mathsf{CPU}}, \mathsf{R/W}, L, \perp)$
```

Figure 19: Description of the Step Circuit

- $\mathsf{Dec}(K, c)$: Parse c as (time, ct). Compute $SK \leftarrow \mathsf{RC.Eval}(K, \mathsf{time})$. Output $\mathsf{SK.Dec}(SK, ct)$.

We note that correctness and encryption with (strongly) constrained key follows directly from the correctness of range constrained PRF and symmetric key encryption scheme. The security property can be easily argued from the security of range constrained PRF and symmetric key encryption. $\square$

D.3 Construction

In this section, we formalize the ideas in Section 5.6 into a full construction of GPRAM.

- $\mathsf{Memory}(1^\lambda, D)$:
  - $K \leftarrow \mathsf{TE.KeyGen}(1^\lambda)$
  - $S \leftarrow \mathsf{PRFKeyGen}(1^\lambda)$ defining $\mathsf{PRF}_S : \{0,1\}^\lambda \mapsto (\{0,1\}^n)^M$
  - $\hat{D} \leftarrow \mathsf{TE.Enc}(K, 0, D)$
  - $D^* \leftarrow \mathsf{OPData}(1^\lambda, \hat{D})$
  - $(\tilde{D}, SK) \leftarrow \mathsf{UGPRAM.Memory}(1^\lambda, D^*)$
  - $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda, 1^{|D|})$
  - $\mathsf{lab}^1_{j,b} \leftarrow \mathsf{PRF}_K(1, j, 0, b), \forall j \in [\lambda], b \in \{0,1\}$
  - $(\mathsf{d}, \hat{D}) \leftarrow \ell OT.\mathsf{Hash}(\mathsf{crs}, D)$
  - Output $\hat{D}$ as the garbled memory and (K, S, SK) as the secret key

- $\mathsf{Program}(SK', i, \Pi)$:
  - Parse $SK' = (K, S, SK)$
  - Let T be the parallel running time of Π
  - $\Pi^* \leftarrow \mathsf{OPProg}(1^\lambda, |D|, 1^T, \Pi)$ where T^* is the parallel running time of Π^*
  - Let $\pi(C)$ be the order for Π^* such that the step circuits in Π^* can be evaluated in

```
order $\pi^{-1}(1), \pi^{-1}(2), ..., \pi^{-1}(T')$ sequentially, where $T'$ is the sequential running time
of $\Pi^*$
  Let $K[i, \tau, 1] = \mathsf{TE.Constrain}(K, (i||\tau||1))$
  $r \leftarrow \mathsf{PRF}_S(i)$
  For $\tau \in [T']$ do
    Let $C = \pi^{-1}(\tau)$
    Let $I$ be the interval of random tape that $C$ needs to access
    Let $C^\tau_{\mathsf{CPU}} := \mathsf{SC}^\tau[i, \tau, K[i, \tau], I, K', r', \pi^{-1}(\tau)]$, where $r' = r_k$ if $C$ is the output
circuit for CPU $k \in [M]$, else $r' = \perp$. The step circuit $\mathsf{SC}$ is described in Figure 19
  Let $\Pi'$ be the PRAM program described by step circuits $\{C^\tau_{\mathsf{CPU}}\}_{\tau \in [T']}$
  Output $\tilde{\Pi} := \mathsf{UGPRAM.Program}(SK, i, \Pi')$
```

- $\mathsf{Input}(SK', i, \{x_k\}_{k \in [M]})$:
```
  Parse $SK' = (K, S, SK)$
  $\hat{x} \leftarrow \mathsf{UGPRAM.Input}(SK, i, \{x_k\}_{k \in [M]})$
  $r \leftarrow \mathsf{PRF}_S(i)$
  Output $\tilde{x} := (\hat{x}, r)$
```

- $\mathsf{Eval}^{\tilde{D}}(i, \mathsf{st}, \tilde{\Pi}, \tilde{x})$:
```
  Parse $\tilde{x} = (\hat{x}, r)$
  $(y, \mathsf{st}') \leftarrow \mathsf{UGRAM.Eval}^{\tilde{D}}(\mathsf{st}, \tilde{P}, \hat{x})$
  Output $y \oplus r$ and $\mathsf{st}'$
```

D.4 Security

For the simulators, the simulated database is simply a garbled database of all zeroes; the simulated program is a garbled stub PRAM program; and the simulated input is a garbled zero string and the correct output XOR the random mask r. To change the garbled program into the simulator indistinguishably, we change the step circuits from $\pi^{-1}(T')$ to $\pi^{-1}(1)$ to their corresponding simulated versions, and finally by the security of timed encryption and the fact that the database and the input was not read by the simulated program at all, we can change the database and the input string to zeroes.

Now to finish the proof, we prove that the change we described is indeed computationally indistinguishable.

Lemma D.5. *Let* Hybrid_t *be the hybrid distribution such that step circuits in* $[t, T']$ *garbles a stub circuit. Assuming the security of adaptive garbled PRAM with unprotected memory access, timed encryption, puncturable PRF, and oblivious PRAM with strong localized randomness, we have* $\mathsf{Hybrid}_t \overset{c}{\approx} \mathsf{Hybrid}_{t-1}$ *for every* $t \in [T']$.

Proof. We prove via a hybrid argument.

- Hybrid_t: This is our starting hybrid.
- $\mathsf{Hybrid}_{t,1}$: In this hybrid, we change how the garbled program and the garbled input are generated. In particular, we will use the simulator for UGPRAM to equivocate the circuit SC^t (UGPRAM will generate some additional step circuits but we do not

need to worry about them in this security proof), and we compute the correct output of SC^t as y_t and send it using UGPRAM's simulator when we are asked to garble the input. The computational indistinguishability follows from the equivocal security of adaptive UGPRAM.

- $\mathsf{Hybrid}_{t,2}$: In this hybrid, we change what is written to the database by y_t. In particular, if it performs a write, we hardwire wData to 0 before encrypting using timed encryption. Note that the adversary now only has constrained key up to $(i, t-1, 1)$, the computational indistinguishability follows from the security of timed encryption scheme.

- $\mathsf{Hybrid}_{t,3}$: In this hybrid, we change how y_t is generated. In particular, we change $\mathsf{state'}$ to 0, and if the circuit is the output circuit for CPU k, we also change the garbled input to $y \oplus r$. Note that the adversary now only has constrained key up to $(i, t-1, 1)$, the computational indistinguishability follows from the security of timed encryption scheme if the circuit is not output, otherwise the distributions are identical.

- $\mathsf{Hybrid}_{t,4}$: By strong localized randomness property of OPRAM, there exists a set $P_t := P_{\Pi^*}(\pi^{-1}(t))$ of size at most $O(1)$ such that property 3 in the definition of strong localized randomness holds. We puncture the PRF key at $I(P_t)$ and I where $I(P_t)$ represents the random tape accessed by the circuits in P_t. We hardwire the PRF outputs for $\pi^{-1}(t)$ and circuits in P_t and use the punctured key in the rest of the step circuits. Now it follows from the correctness property of puncturable PRF that programs garbled in $\mathsf{Hybrid}_{t,4}$ and $\mathsf{Hybrid}_{t,5}$ have the same output in each step circuit for each input x, therefore by adaptive security of UGPRAM, $\mathsf{Hybrid}_{t,3}$ is computationally indistinguishable from $\mathsf{Hybrid}_{t,4}$.

- $\mathsf{Hybrid}_{t,5}$: In this hybrid, we replace the hardwired PRF outputs with random strings. The computational indistinguishability follows from the security of puncturable PRFs.

- $\mathsf{Hybrid}_{t,6}$: In this hybrid, we use the simulator for UGPRAM to equivocate all the circuits in P_t as well. The computational indistinguishability follows from the equivocal security of adaptive UGPRAM.

- $\mathsf{Hybrid}_{t,7}$: In this hybrid, we change the memory location L accessed by SC^{t-1} to be sampled according to the procedure of OPRAM but independently of everything else. The computational indistinguishability follows from the security of strong localized randomness property of OPRAM.

- $\mathsf{Hybrid}_{t,8}$: In this hybrid, we change SC^t to a dummy circuit that outputs y_t (which is a string because it does not depend on the input, furthermore, the encryption is done using a strongly constrained key), and the rest of the circuits in P_t is reverted to the circuits used in $\mathsf{Hybrid}_{t,5}$. y_t does not change due to the encrypting with strongly constrained key property of timed-encryption, and therefore the computational indistinguishability follows from the equivocal security of adaptive UGPRAM.

- $\mathsf{Hybrid}_{t,9}$: In this hybrid, we reverse the change made in $\mathsf{Hybrid}_{t,5}$, namely, we replace the hardwired random strings in the step circuits in P_t with the corresponding PRF output. The computational indistinguishability follows from the security of puncturable PRFs.

- Hybrid_{t-1}: The difference between this hybrid with $\mathsf{Hybrid}_{t,9}$ is the punctured PRF keys. In this hybrid, we reverse the change made in $\mathsf{Hybrid}_{t,4}$. Similar to before, the computational indistinguishability follows from the adaptive security of UGPRAM.

This completes the proof of the lemma. $\square$