Adaptively Secure Garbling Schemes for Parallel Computations

Kai-Min Chung[1] and Luowen Qian[2(✉)]

[1] Institute of Information Science, Academia Sinica, Taipei, Taiwan
kmchung@iis.sinica.edu.tw
[2] Boston University, MA, USA
luowenq@bu.edu

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.

Kai-Min Chung is 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.

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 $\tilde{C}$ 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 $\tilde{C}$. In particular, for the circuit settings, we refer to the size of $\tilde{C}$ 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 $\tilde{D}$, and having executed some garbled programs on the database and obtained

[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.

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 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. *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 construction for circuit formally in Section 4. Please see the full version of our paper for the construction 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.

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. 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.

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.

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] 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.

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.

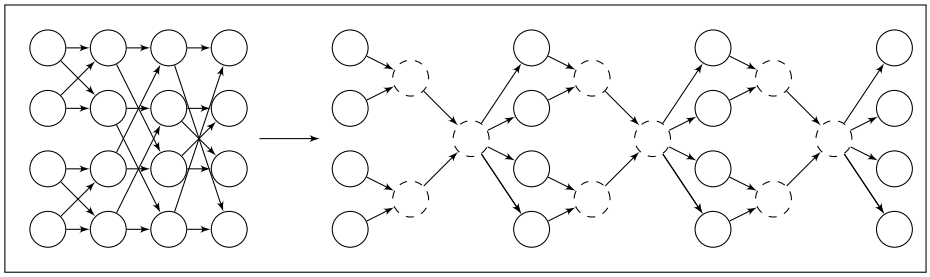

Fig. 1. Transforming a toy sorting network using "parallel checkpoints." The undashed vertices corresponds to the step circuits that do the actual sorting.

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, please refer to the full version.

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 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 $\bar{c}$.*
- $\bar{m} \leftarrow \mathsf{Dec}(\mathsf{key}, \bar{c})$*: 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* $\bar{c} \leftarrow \mathsf{Enc}(\mathsf{key}, \bar{m})$*.*

$\mathsf{SimEncExpt}^{b\in\{0,1\}}(1^\lambda, \mathcal{A})$
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 $\mathcal{A}_2(\mathsf{st}', \mathsf{key})$

Fig. 2. Simulated Encryption Experiment

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 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.

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

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

3.5 Laconic Oblivious Transfer

Definition 2 ([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* $\hat{\mathsf{D}}$.
- $\mathsf{e} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$. *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.

– ***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.*

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

Fig. 3. Sender Privacy Security Game

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

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.*

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

Fig. 4. Sender Privacy Security Game for Writes

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

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

Remark 1. 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 4 ([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. *An adaptive garbling scheme for circuits is a tuple of PPT algorithms*
$(\mathsf{AdaGCircuit}, \mathsf{AdaGInput}, \mathsf{AdaEval})$ *such that:*

- $(\tilde{C}, \mathsf{st}) \leftarrow \mathsf{AdaGCircuit}\,(1^\lambda, C)$*. 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 $(\tilde{C}, \mathsf{st}) \leftarrow \mathsf{AdaGCircuit}\,(1^\lambda, C)$ *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.*

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.

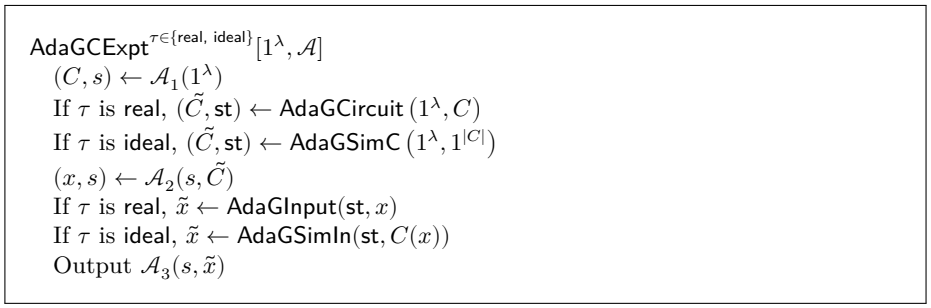

Fig. 5. Adaptive Security Game of Adaptive Garble Circuits

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 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 ℓOT 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 d of database $\{b_k\}_{k\in[2^M]}$
  Reinterpret the ℓOT 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 Label

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

Fig. 6. Block-writing security simulator

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

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:

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

Fig. 7. Hashing Sub-Circuit

- $\mathsf{GHash}(1^\lambda, H, \{\mathsf{Key}_i\}_{i\in[|D|]}, \{\mathsf{Key}'_i\}_{i\in[\lambda]})$: 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 $\{\mathsf{Keys}_i\}_{i=2,\ldots,|D|/\lambda-1}$
  $\{\mathsf{Keys}_i\}_{i=|D|/\lambda,\ldots,2|D|/\lambda-1} \leftarrow \{\mathsf{Key}_i\}_{i\in[|D|]}$
  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 $\{\mathsf{lab}_i\}_{i\in[|D|]}$
  $\{\mathsf{Label}_i\}_{i=|D|/\lambda,\ldots,2|D|/\lambda-1} \leftarrow \{\mathsf{lab}_i\}_{i\in[|D|]}$
  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 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 $\left\{\tilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}, \tilde{\mathsf{C}}_{i,j}^{\mathsf{hash}}, \tilde{\mathsf{C}}_{i}^{\mathsf{write}}\right\}$, 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 $d \leq \ell \leq |C|$, 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 $\tilde{D}$ 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 $r \leftarrow \{0,1\}^{M\ell}$

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: d

If τ is 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 τ is 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 $\mathsf{Send}(\mathsf{crs}, \mathsf{d}, i, f_0, f_1)$

Fig. 8. Description of the Evaluation Circuit

For $i = 1$ to ℓ do:

Let $C_{g,1}, C_{g,2}$ denote the two input gates of gate g

$$\tilde{\mathsf{C}}_{i,k}^{\mathsf{eval}} \leftarrow \mathsf{GCircuit}(1^\lambda, \mathsf{C}_{\mathsf{eval}}^{\mathsf{real}}[\mathsf{crs}, C_{P_{i,k},1}, C_{P_{i,k},2}, P_{i,k}, (r_{C_{P_{i,k},1}}, r_{C_{P_{i,k},2}}, r_{P_{i,k}}), \mathsf{PRF}_K(1,i,k,0), \mathsf{PRF}_K(1,i,k,1)], \{\mathsf{PRF}_K(0,i,j,b)\}_{j\in[\lambda],b\in\{0,1\}})$$

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 ℓ do:

Let $C_{g,1}, C_{g,2}$ denote the two input gates of gate g

$\mathsf{e} \leftarrow \mathsf{GCEval}(\tilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}, \{\mathsf{lab}_{0,j}\}_{j\in[\lambda]})$

$\mathsf{e} \leftarrow \ell OT.\mathsf{Receive}^{\tilde{D}}(\mathsf{crs}, \mathsf{e}, C_{P_{i,k},1})$

$(\gamma_k, \mathsf{lab}_{1,k}) \leftarrow \ell OT.\mathsf{Receive}^{\hat{D}}(\mathsf{crs}, \mathsf{e}, C_{P_{i,k},2})$

$\{\mathsf{lab}_{2,j}\}_{j\in[\lambda]} \leftarrow \mathsf{GHEval}(\{\tilde{\mathsf{C}}_{i,j}^{\mathsf{hash}}\}_{j\in[M-1]}, \{\mathsf{lab}_{1,k}\}_{k\in[M]})$

$\mathsf{e} \leftarrow \mathsf{GCEval}(\tilde{\mathsf{C}}_i^{\mathsf{write}}, \{\mathsf{lab}_{2,j}\}_{j\in[\lambda]})$

$\{\mathsf{lab}_{0,j}\}_{j\in[\lambda]} \leftarrow \ell OT.\mathsf{ReceiveWriteBlock}^{\tilde{\mathsf{D}}}(\mathsf{crs}, i, \{\gamma_k\}_{k\in[M]}, \mathsf{e})$

Recover the contents of the memory D from the final state $\hat{D}$

Output $D_{N-m+1} \oplus r_{N-m+1}||...||D_N \oplus r_N$

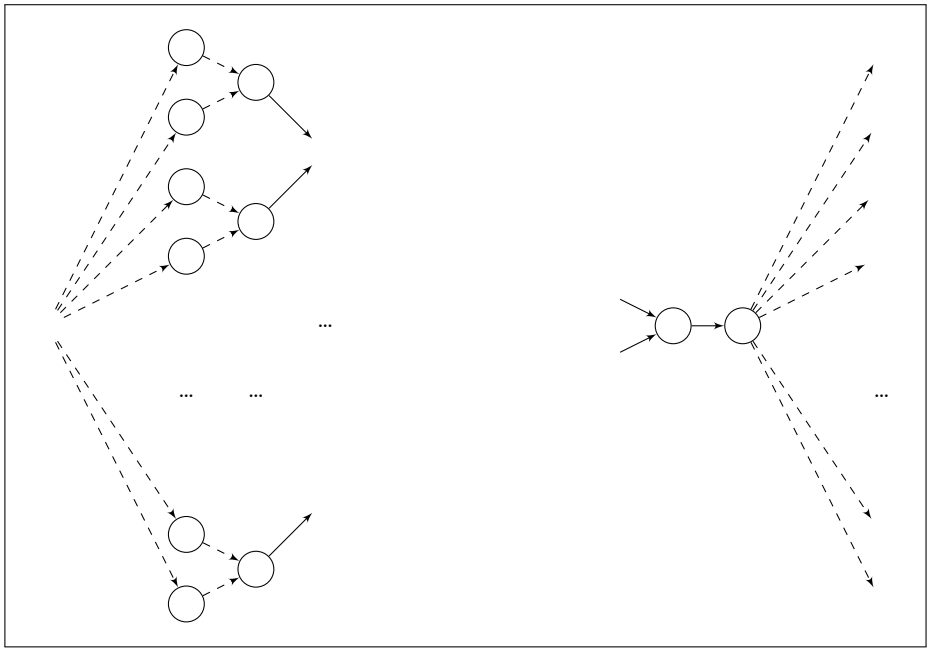

Fig. 9. Illustration of the pebbling graph for one layer: $\widetilde{\mathsf{C}}_{i,k}^{\mathsf{eval}}$ are leaf nodes, $\widetilde{\mathsf{C}}_{i,j}^{\mathsf{hash}}$ are intermediate nodes and the root node, finally $\widetilde{\mathsf{C}}_i^{\mathsf{write}}$ is the extra node at the end. Dotted edges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

Communcation Complexity of AdaGInput. It follows from the construction that the communication complexity of 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 AdaGInput. The running time of AdaGInput grows linearly with $|C|$. However, it's possible to delegate the hashing of zeros to the

offline phase, i.e. AdaGCircuit. In that case, the running time only grows linearly with $n + \log |C|$.

Parallel Efficiency. With a good Partition algorithm and number of processors as many as the width of the circuit, 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 the full version.

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.

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.

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.

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.

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.

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.