Privacy Preserving Computation in Cloud Using Reusable Garbled Oblivious RAMs

Yongge Wang[1] and Qutaibah M. Malluhi[2]

[1] UNC Charlotte, 9201 University City Blvd., NC 28223, USA yonwang@uncc.edu
[2] Qatar University, Doha, Qatar qmalluhi@qu.edu.qa

Abstract. When users store encrypted data in a cloud environment, it is important for users to ask cloud to carry out some computation on the remote data remotely. ORAM is a good potential approach to carry out this kind of remote operation. In order to use ORAM for this purpose, we still need to have garbled programs to run on ORAM. Goldwasser et al and Lu-Ostrovsky initiated the study of garbled RAM machines in their 2013 Crypto papers. Goldwasser et al's scheme is based on fully homomorphic encryption schemes and attribute based encryption schemes for general RAM machines. Lu and Ostrovsky's scheme is based on one-time garbled circuits and for each input, one has to design as many one-time garbled circuits as ORAM CPU running steps. That is, for each execution of the program, the data owner needs to upload a new program to the cloud to run on ORAM. Using recent results on indistinguishability obfuscation, this paper designs alternative reusable garbled ORAM programs. The reusable garbled ORAM CPU constructed in this paper is of constant size while the size of the garbled ORAM CPUs by Lu and Ostrovsky depends on the number of ORAM CPU running steps.

1 Introduction

Cloud computing techniques become more and more popular and users begin to store their private encrypted data in cloud services. In order to take full advantage of the cloud computing paradigm, it is important to design efficient techniques to carry out computation over encrypted data in the cloud without downloading the data to a local machine. Though computation over encrypted data helps to protect the privacy of the data, it does not hide the access pattern to data. A natural solution is to use oblivious RAM techniques by Goldreich and Ostrovsky [12] to carry out computation over encrypted data, which provably hides all access patterns.

In order to use ORAM schemes, a trusted CPU is required. Since users may not trust the CPU powers at cloud environments, it has been recommended for the user to run the trusted CPU at client site and to treat the cloud as a large random access memory storage service. The disadvantage of this approach is the heavy communication overhead between the client and the cloud. For example, the most efficient ORAM scheme requires at least $O(\log^2 n)$ memory accesses for each individual memory access, where the cloud database contains n unit blocks of data.

Lu and Ostrovsky [24] and Goldwasser et al [13] initiated an alternative approach to let the cloud run a garbled version of the ORAM CPU. In this approach, the client

machine only needs to submit the garbled ORAM CPU to the cloud and the cloud only needs to return the encrypted outputs to the client. Thus the communication overhead could be significantly reduced in case the cloud database size is large. One disadvantage for Lu and Ostrovsky's approach [24] is that their garbled RAM CPU is not succinct and can be used only for one time. For example, if the ORAM CPU runs t-steps for one input x, then the garbled ORAM CPU for the input x is at the size of $O(t)$. Lu and Ostrovsky [24] lists it as a tempting open problem to use Goldwasser et al's [14] reusable garbled circuits to design reusable garbled RAMs. It should be noted that Goldwasser et al [13] designed reusable garbled RAM machines using fully homomorphic encryption (FHE) and Attribute Based Encryption schemes for RAM machines.

In recent years, several indistinguishability obfuscation schemes have been designed (see, e.g., Jain-Lin-Sahai [18]). By converting the ORAM CPU to an NC^1 circuit and then using obfuscation schemes, this paper designs practical reusable garbled ORAMs for cloud computation over encrypted data. Our scheme is succinct since the garbled ORAM CPU program is of constant size. Furthermore, for commonly used cloud application programs, they are encrypted and stored in the database server together with user data. Thus for each execution of the program over the encrypted database (e.g., a database search query), the user only needs to submit an encrypted keyword to the server, where the encrypted keyword is approximately the same size as the keyword. In a summary, the contributions of this paper are two-folds. First, this paper presents an alternative garbled ORAM program design which is different from Lu-Ostrovsky [24] and Goldwasser et al [13]. Secondly, the garbled ORAM programs in this paper are reusable while the scheme in [24] is not reusable.

We close this section by introducing some notations. We use κ to denote the security parameter. A function f is said to be negligible in an input parameter κ if there exists κ_0 such that for all $\kappa > \kappa_0$, $f(\kappa) < \kappa^{-n}$ for all $n > 0$. For convenience, we write $f(\kappa) = \text{negl}(\kappa)$. Two ensembles, $X = \{X_\kappa\}_{\kappa\in N}$ and $Y = \{Y_\kappa\}_{\kappa\in N}$, are said to be computationally indistinguishable if for all probabilistic polynomial-time algorithm D, we have $|Pr[D(X_\kappa, 1^\kappa) = 1] - Pr[D(Y_\kappa, 1^\kappa) = 1]| = \text{negl}(\kappa)$.

The structure of this paper is as follows. Section 2 provides a background discussion and reviews necessary techniques required for the construction of garbled ORAMs in this paper. Our main construction of reusable garbled ORAMs is presented in Section 3.

2 Cloud data storage and oblivious RAMs

Cloud storage systems may be interpreted as databases stored at the cloud servers, There have been extensive research on public and private databases in the literature. In the public database setting, the database is published and individual users need to retrieve some entries from the database without letting the database server know which entry it has retrieved. A straightforward solution is to let users to download the whole database though it is not practical. To address this challenge, Chor, Goldreich, Kushilevitz, and Sudan [6] introduced the private information retrieval (PIR) concept in an information theoretic setting. PIR protocol makes it possible for users to obtain information from a database without downloading the whole database. At the same time, PIR protocol will

not reveal to the database server which entry the user has retrieved. In an extended PIR protocol [6], one could have many copies of the identical database without allowing them to communicate with each other. Chor and Gilboa [5], Ostrovsky and Shoup [26], and others considered the computational PIR, in which the database is restricted to perform polynomial time computations. A single database based PIR was constructed by Kushilevitz and Ostrovsky [22] assuming that certain public-key encryption scheme exists. Since then, several single database PIR schemes with better bounds have been proposed and studied. For a brief survey, it is referred to Ostrovsky and Skeith [27]. Though PIR techniques find important applications in many domains, it is not sufficient to address the challenges in the privacy preserving cloud data distribution systems that we are facing.

In the private database setting, users upload private databases to a remote database server while keeping the database private from the remote database administrators. At a later time, users should be able to search and retrieve entries with certain keyword from the remote database. Based on the physically shielded Central Processing Unit (CPU) technique [20], Goldreich and Ostrovsky [12] proposed a theoretical treatment of software protection by formulating the problem in the setting of learning a program structure by observing its execution. Using this new formulation, they reduced this problem to on-line simulation of any programs on oblivious RAMs (random access machines). A machine is oblivious if its accesses to memory locations are independent of the input values with the same running time. We may apply these schemes in the cloud computing environments (e.g., search over encrypted texts) as follows: the physically shielded CPU is interpreted as the user at the client side and the memory locations are interpreted as the cloud storage. Though the scheme in [11,12,25] is asymptotically efficient and nearly optimal, it is inefficient in practice with large hidden constants in the big-O notation and a heavy communication overhead between the client and the server.

In the RAM (random access machine) model, the CPU performs basic arithmetic, logical, control and input/output operations specified by the instructions. The CPU can be considered as a stateful processor where the state Σ is determined by the content in the registers. The registers store program counters, query counters, session information, cryptographic keys, and other information. Among these registers, there is an accumulator where intermediate arithmetic and logic results are stored. Throughout this paper, we will assume that CPU could perform the following operations:

1. Perform arithmetic instructions $+, -, \times, \lfloor x/y \rfloor$. For each arithmetic operation $f(x, y)$, there are two inputs x and y. The value of x should be already in the accumulator and y should be a value in the memory cell to be fetched.
2. Generate a random number and put it in the accumulator.
3. Read data from a memory cell to the accumulator and write the value in the accumulator to a memory cell. Note that this kind of operations will include the user data inputs and outputs if we use some fixed memory cells for user inputs and some other fixed memory cells for user outputs.
4. Control transfer instructions: “GOTO X”, “IF $X = 0$ THEN GOTO Y”, and “IF $X > 0$ THEN GOTO Y”.
5. HALT: terminates the execution of the program.

During the execution of the RAM CPU, each read/write operation to memory cells could be viewed as a query (op, v, x) where *op* equals to READ or WRITE, v is the data identifier and x is the value. Without loss of generality, we always assume that (op, v, x) is contained in a register that is called an interface register. In the RAM machine model, the actual programs are stored in the memory cells. Thus RAM CPU can be considered as a universal machine that reads programs from the memory cell and executes the instructions step by step. Based on this interpretation, we will not distinguish data and programs throughout this paper.

In order to protect the memory cell access patterns of the RAM CPU, the client holds a secret key for a semantically secure probabilistic encryption scheme. The data and programs uploaded to RAM memory cells are encrypted using the secret key. The clients stores n blocks of data (v_i, x_i) where v_i is the data identifier or location-index and x_i is the data payload. By default, the data block (v_i, x_i) is stored at physical location i in the memory cell. As we have discussed in the previous paragraphs, the RAM CPU interacts with data stored in the memory cells by issuing commands "READ (v_i, x_i)" and "WRITE (v_i, x_i)". By default, the RAM machine does not hide the fact that the CPU has accessed the data stored at the physical position i (by default, it is (v_i, x_i)) even if the data payload (v_i, x_i) itself is encrypted and remains perfectly secure. In order to hide the actual data blocks that the client accessed, the oblivious RAM (ORAM) machine is introduced where the data block (v_i, x_i) is no longer stored at the physical position i. Instead, a random permutation is used to store (v_i, x_i) at a random location. In order to hide the event that one data block is accessed for multiple times, further mechanisms (e.g., a cache) are used. Several commonly used constructions of oblivious RAMs are presented in next sections. The security for ORAMs is expressed in the following definition which is based on [12,29,16].

Definition 1. *Assume that the client store a sequence of data blocks $X = \{(v_1, x_1), \cdots, (v_n, x_n)\}$ at the server. Each data block (v_i, x_i) is located at a physical location $\pi(i)$. The client (or the ORAM CPU) issues a sequence of operations $(op_1, a_1, y_1), \cdots, (op_m, a_m, y_m)$ to the server where each (op_i, a_i, y_i) represents a read or write command. For example, a command $(READ, a_i, y_i)$ asks the server to read the content at the physical location a_i to the variable y_i. The sequence of operations $(op_1, a_1, y_1), \cdots, (op_m, a_m, y_m)$ is called an access pattern $\mathcal{A}(X)$ on client data blocks X. An oblivious RAM machine is* secure *if for any two data blocks X and Y of equal length, the access patterns $\mathcal{A}(X)$ and $\mathcal{A}(X)$ are computationally indistinguishable for any one but the client who holds the secret key.*

The first oblivious RAM simulation was designed by Goldreich [11] using the "square root" construction. For a RAM machine with n memory cells denoted by an array $R[1..n]$, an oblivious RAM with a memory array $OR[1..n + 2\sqrt{n}]$ was designed in [11]. The portion $OR[n + \sqrt{n} + 1..n + 2\sqrt{n}]$ of size $\sqrt{n}$ is used by the ORAM as the cache space (or a shelter). For the first $n + \sqrt{n}$ cells, choose a random permutation

$$\pi : \{1, \cdots, n + \sqrt{n}\} \rightarrow \{1, \cdots, n + \sqrt{n}\}$$

and let $OR[\pi(i)] = R[i] = (v_i, x_i)$, where we assume that $R[i]$ contains a dummy value for $n < i \leq n + \sqrt{n}$. Each time when the ORAM accesses a data block (v_i, x_i) from

$OR[\pi(i)] = R[i]$, it stores this value (v_i, x_i) in the cache $OR[n + \sqrt{n} + 1..n + 2\sqrt{n}]$. For each new query of a data block (v_j, x_j), ORAM checks all values in $OR[n+\sqrt{n}+1..n+2\sqrt{n}]$ to see whether (v_j, x_j) has been cached there already. If the data block is found, ORAM only needs to make a dummy access to another cell $OR[\pi(n+l)]$ where l is the counter. That is, this is the l-th dummy memory cell access. If the data block is not found, ORAM loads the data block (v_j, x_j) from $OR[\pi(j)]$ directly. After $\sqrt{n}$ memory cell accesses, ORAM needs to re-shuffle data blocks in the memory cells using an oblivious sorting process.

Goldreich [11] used Batcher's sorting network which takes $O(n \log^2 n)$ memory cell accesses. Thus the amortized memory access is $O(\sqrt{n} \log^2 n)$ which is significantly larger. In a summary, each memory access of the RAM is translated to $O(\log n)$ square-root ORAM memory accesses (without re-shuffling) and amortized $O(\sqrt{n} \log^2 n)$ ORAM memory accesses (including re-shuffling cost).

Hierarchical ORAM. Goldreich and Ostrovsky [12] introduced a more efficient hierarchical construction of oblivious RAM machines. In this approach, a tree is constructed. Each tree node is called a bucket which contains $\log n$ cells. Each level of the tree is called a buffer. The first buffer (including $T_{1,1}$ and $T_{1,2}$) at the root level is used as the cache space (or a shelter) as in the "square root" approach. The i-th level buffer has 2^i nodes. Each bucket will hold only one data block on average. At the beginning, the tree has $L = \log n$ levels and the data blocks in $R[1..n]$ of the original RAM are stored in the nodes $T_{L,1}, \cdots, T_{L,2^L}$ where $2^L = n$. Recent works by Pinkas-Reiman [29] and Goodrich-Mitzenmacher [16] show that if cuckoo hashing [28] is used, then each bucket (that is, the tree node $T_{i,j}$) only needs to contain 4 memory cells to guarantee that the i-level can hold 2^i data blocks. Note that in the i-th buffer, we have $4 \cdot 2^i$ memory spaces. This is sufficient to store 2^i RAM's data blocks and 2^i dummy values using the cuckoo hashing. In Goodrich-Mitzenmacher [16] simulation, a further $O(\log n)$-size cache Q and a shared stash are included to improve the performance.

To query the data block (v_i, x_i) that were supposed to be stored at $R[i]$ of the original RAM machine, the ORAM first scans the cache Q, the shared stash, and the entire first level buffer (that is, the buckets $T_{1,1}$ and $T_{1,2}$) to check whether (v_i, x_i) has already been accessed or not. If the data block (v_i, x_i) is found, ORAM continues to read a dummy node from each buffer. If (v_i, x_i) is not found in the cache, ORAM checks whether $T_{2,h_1(i)}$ or $T_{2,h_2(i)}$ contains (v_i, x_i). Similarly, if the data block (v_i, x_i) is found at the second buffer, ORAM continues to read one dummy node from each lower buffer. If (v_i, x_i) is not found in the second buffer, it continues to look for it in lower buffers. After the cache or the i-th buffer is full, oblivious sorting is used to move the cache to the second buffer or to move the i-th buffer to the $(i + 1)$-th buffer. The randomized Shell sorting algorithm [15] could be used to sort the buffers with $O(n \log n)$ memory cell accesses.

In the hierarchical ORAM simulation, each RAM memory access is replaced by $O(\log n)$ ORAM memory accesses. For the randomized Shell sorting algorithm based re-shuffling of level i, it takes $O(2^i \log 2^i)$ memory accesses. Thus the amortized memory access is $O(\log^2 n)$. Note that it is still quite slow to re-shuffle a level of the tree. For example, it takes $O(n \log n)$ memory accesses to re-shuffle the lowest level.

Unconditionally secure ORAM. The ORAM models that we have discussed in the previous paragraphs use oracles (or cryptographic pseudorandom functions) to shuffle the memory cells of the original RAM. If we use flipped coins to shuffle the memory cells, then we need to keep a track of the flipped coins. Recently, Damgård et al [7] and Ajtai [1] achieved this by storing the flipped coins in the additional memory cells. Using these recorded flipped coins, one can construct ORAMs without the use of random oracles. Note that the approach in Ajtai [1] has a small probability that ORAMs will fail on some inputs. The approach in [7] does not have this issue. Furthermore, Ajtai [1] showed that the shielded CPU requirements in the ORAM model could be dropped. From a first reading, the reader may interpret Ajtai's ORAM as a obfuscated program or a reusable garbled circuit. Thus there is a "conflict" with the impossibility results for code obfuscation [1]. However, it should be noted Ajtai's ORAMs behave more like oblivious Turing machines (see, Pippenger and Fischer [30]) where the focus is on hiding memory access pattern instead of content confidentiality. Furthermore, it should be noted that the ORAM model requires memory cells to be encrypted while the code obfuscation scheme does not require encrypted memory cells[3].
ORAM with $O(\sqrt{n})$ local storage. In the previous paragraphs, we discussed ORAM construction with constant local storage. The cost of this restriction is the external storage expansion and communication overhead. For example, in Pinkas and Reinman's construction [29], we need to use $8n$ to $12n$-bits storage size for n-bits data and each data access requires $O(\log^2 n)$ round trips to the external storage. In Goldreich and Ostrovsky's [12] hierarchical construction, one needs $O(n \log n)$-unit data storage for n-unit data and each data access requires $O(\log^3 n)$ round trips to the external storage. By requiring $O(\sqrt{n})$-unit storage at client side. Boneh, et al [4] introduced an ORAM/OS (ORAM based oblivious storage) scheme that only requires around 5 round trips for each data access and the external storage size is slightly large than n-unit. In the construction of [4], it uses the square root algorithm by converting the $\sqrt{n}$-unit cache to a hierarchy tree and by keeping a map copy (only addresses but not data part) of the hierarchy tree at the client side. The construction in [4] is a hybrid construction of the square root algorithm [11] and the optimized Hierarchical algorithm [29].

2.1 Techniques for efficient ORAMs

Cuckoo hashing and the randomized data-oblivious Shell sorting algorithm are used to improved the ORAM performance. Cuckoo hashing was introduced by Pagh and Rodler [28] as a hash table with a worst case constant lookup time. In order to store n elements from the domain U, we use two tables (i.e., arrays) $T_1[1..n]$ and $T_2[1..n]$, each of them can store n elements. We also use two hash functions $h_1, h_2 : U \rightarrow \{1, \cdots, n\}$. Each element $x \in U$ is stored in either $T[h_1(x)]$ or $T[h_2(x)]$, but never in both. It is clear that for each x, the position of x could be found in 2 steps.

In order to insert an element into the arrays, the following process is taken: if $T_1[h_1(x)]$ is empty, place it here and stop. If $T_1[h_1(x)]$ is not empty, then let $y = T_1[h_1(x)]$ and store x here. If $T_2[h_2(y)]$ is empty, place y here and stop. If $T_2[h_2(y)]$ is not empty, then let $z = T_2[h_2(y)]$, store y here, and continue to insert z into the arrays.

[3] The first author would like to thank Ajtai for pointing out this fact to the authors.

If the insertion process does not stop after a pre-defined maximum number of steps, choose two new hash functions to start from the beginning.

To improve the robustness of cuckoo hashing, Kirsch et al [21] used a stash as an additional space to hold elements that would cause failures for the cuckoo hashing process. With a stash, the process fails if the stash itself overflows.

The randomized and data-oblivious version of Shellsort algorithm [32] was introduced by Goodrich [15]. It runs in $O(n \log n)$ time and succeeds with a very high probability. For an unsorted array $T[1..n]$ of n elements, the Shellsort algorithm works by choosing an offset sequence $(o_1, \cdots, o_p)$ with $o_i < n$. For each $i < p$ and $j \leq o_i$, sort the sub-array $T[j, j + o_i, j + 2o_i, \cdots]$ using insertion-sort. Note that Leighton and Plaxton [23] also studied randomized data-oblivious sorting algorithms with average $O(n \log n)$ time complexity.

Note that in the data-dependent insertion sort, one starts from the second element, insert the scanned element in the correct position and then do the same for the third element and so on. The performance of the Shellsort algorithm depends on the choice of the offset sequence. For example, Pratt [31] chooses the offset sequence as all products of powers of 2 and 3 to obtain a worst-case running time of $O(n \log^2 n)$.

For Goodrich's randomized data oblivious Shellsort [15], let $T[1..n]$ be the array that we want to sort and assume that n is a power of 2. Set the offset sequence $o_i = n/2^i$ for $i = 1, \cdots, \log n$. For each offset o_i, the array T is partitioned into regions $T[1..o_i]$, $T[o_i + 1..2o_i]$, $\cdots$. A compare-exchange operation on two regions T_1 and T_2 is defined as follows: let $\pi : \{1, \cdots, o_i\} \to \{1, \cdots, o_i\}$ be a random permutation. For $j = 1, \cdots, o_i$, compare and exchange the values $T_1[j]$ and $T_2[\pi(j)]$ according to their order. Note that if we take the permutation π as the identity permutation, then this is the same as the Shellsort. The Goodrich's randomized data oblivious Shellsort can now be described as follows. For each $o_i = n/2^i$ starting from $i = 1$, do

- divide T into regions T_1, T_2, $\cdots$, T_{n/o_i};
- do a shaker pass on $T_1, T_2, \cdots, T_{n/o_i}$, where increasing sequence and decreasing sequence of adjacent-regions are compared.
  - Region compare-exchange T_i and T_{i+1}, for $i = 1$, 2, $\cdots$, $n/o_i - 1$
  - Region compare-exchange T_{i+1} and T_i, for $i = n/o_i - 1$, $\cdots$, 2, 1.
- do an extended brick pass on $T_1, T_2, \cdots, T_{n/o_i}$, where regions that are 3 offsets apart, 2 offsets apart, odd-even adjacent, and even-odd adjacent are compared.
  - Region compare-exchange T_i and T_{i+3}, for $i = 1$, 2, $\cdots$, $n/o_i - 3$
  - Region compare-exchange T_i and T_{i+2}, for $i = 1$, 2, $\cdots$, $n/o_i - 2$
  - Region compare-exchange T_i and T_{i+1}, for even $i = 2, \cdots, n/o_i - 1$
  - Region compare-exchange T_i and T_{i+1}, for odd $i = 1, \cdots, n/o_i - 1$

3 Reusable garbled ORAMs

Lu and Ostrovsky [24] showed how to design one-time non-reusable garbled ORAMs by constructing t pairs of garbled circuits $(\mathcal{O}^i_{ORAM}, \mathcal{O}^i_{CPU})$ for $i = 1, \cdots, t$, where t is the maximum runtime of the ORAM, $\mathcal{O}^i_{ORAM}$ simulates the ith-step memory read-/write command, and $\mathcal{O}^i_{CPU}$ simulates the ith-step shielded CPU operation. Gentry et al [10] showed that in order to prove the security for the garbled RAM scheme in [24],

an additoinal circularity assumption is required. Gentry et al [10] then proposed two new constructions to avoid this additional assumption. In this section, we present our construction of practical reusable garbled ORAMs which is based on secure Indistinguishability obfuscation schemes.

3.1 Constrained peseudo random functions

In order to avoid the circularity assumption, Gentry et al [10] used a concept of revocable PRFs: Let $G : \{0,1\}^s \to \{0,1\}^{2s}$ be a pseudorandom generator and we can write $G_0(x)$ to denote the left half of the output $G(x)$ and $G_1(x)$ to denote the right half of the output $G(x)$. That is, $G(x) = G_0(x)||G_1(x)$. For any key $k \in \{0,1\}^s$ and input $x \in \{0,1\}^n$, the pseudorandom fucntion is defined as $F_k(x) = G_{x[n-1]}(\cdots(G_{x[0]}(k))\cdots)$.

A constrained (or revocable) pseudorandom function is defined in such a way that given the description of a constrained pseudorandom function, one cannot compute the output of the pseudorandom function for some excluded inputs. This can be easily achieved using GGM-pseudorandom functions. For example, if we want to exclude the input 0^n, instead of giving out the key k, we can give the following description of the pseudorandom function

$$F_k : \{G_1(k), G_0(G_1(k)), \cdots, G_1(G_0(G_0(\cdots(G_0(k))\cdots)))\}.$$

Goldwasser et al [13] designed reusable garbled RAMs using fully homomorphic encryption (FHE) schemes and attribute based encryption (ABE) schemes for RAMs. As we have mentioned in previous sections, these schemes are neither efficient nor secure against active adversaries.

3.2 Garbled Circuits (GC)

We first briefly review the formal definition of garbled circuits and related concepts.

Definition 2. *A functional encryption scheme* $\texttt{FE}$ *for a class of functions* $\{\mathcal{F}_n\}_{n\in N}$ *is a tuple of probabilistic polynomial time algorithms* ($\texttt{FE.Setup}, \texttt{FE.KeyGen}, \texttt{FE.Enc}$, $\texttt{FE.Dec}$) *with the following properties*

- $(\texttt{fmpk}, \texttt{fmsk}) = \texttt{FE.Setup}(1^\kappa)$ *outputs a master public key* $\texttt{fmpk}$ *and a master secret key* $\texttt{fmsk}$ *on the security parameter* κ*.*
- $\texttt{fsk}_f = \texttt{FE.KeyGen}(\texttt{fmsk}, f)$ *outputs a secret key for a function* f*.*
- $c = \texttt{FE.Enc}(\texttt{fmpk}, x)$ *outputs a ciphertext for* x*.*
- $y = \texttt{FE.Dec}(\texttt{fsk}_f, c)$ *outputs the value* y *which should equal* $f(x)$*.*

The functional encryption scheme is correct if $y \neq f(x)$ *with a negligible probability.*

The security of functional encryption scheme requires that an adversary learns nothing about the input x other than the output $f(x)$.

Definition 3. *(FE security) Let* $\texttt{FE}$ *be a functional encryption scheme for a family of functions* $\mathcal{F} = \{\mathcal{F}_n\}_{n\in N}$*. For a pair of probabilistic polynomial time algorithms* $A =$

(A_0, A_1) *and a probabilistic polynomial time simulator S, define two experiments:*

$\underline{\mathtt{Exp}^{\mathtt{real}}_{\mathtt{FE},A}(1^\kappa):}$	$\underline{\mathtt{Exp}^{\mathtt{ideal}}_{\mathtt{FE},A,S}(1^\kappa):}$
$(\mathtt{fmpk}, \mathtt{fmsk}) \leftarrow \mathtt{FE.Setup}(1^\kappa)$	$(\mathtt{fmpk}, \mathtt{fmsk}) \leftarrow \mathtt{FE.Setup}(1^\kappa)$
$(f, \mathtt{state}_A) \leftarrow A_1(\mathtt{fmpk})$	$(f, \mathtt{state}_A) \leftarrow A_1(\mathtt{fmpk})$
$\mathtt{fsk}_f \leftarrow \mathtt{FE.KeyGen}(\mathtt{fmsk}, f)$	$\mathtt{fsk}_f \leftarrow \mathtt{FE.KeyGen}(\mathtt{fmsk}, f)$
$(x, \mathtt{state}'_A) \leftarrow A_2(\mathtt{state}_A, \mathtt{fsk}_f)$	$(x, \mathtt{state}'_A) \leftarrow A_2(\mathtt{state}_A, \mathtt{fsk}_f)$
$c \leftarrow \mathtt{FE.Enc}(\mathtt{fmk}, x)$	$\bar{c} \leftarrow S(\mathtt{fmpk}, \mathtt{fsk}_f, f, f(x), 1^{\lvert x \rvert})$
$\mathrm{output}(\mathtt{state}'_\mathtt{A}, \mathtt{c})$	$\mathrm{output}(\mathtt{state}'_\mathtt{A}, \bar{\mathtt{c}})$

The scheme is said to be (single-key) secure in the full simulation security model if there exists a probabilistic polynomial time simulator S such that for all pairs of probabilistic polynomial time adversaries (A_0, A_1), the outcomes of the two experiments are computationally indistinguishable.

Definition 4. *Let $\mathcal{C} = \{\mathcal{C}_n\}_{n \in N}$ be a family of circuits such that C_n is a set of boolean circuits that take n-bit inputs. A garbling scheme for $\mathcal{C}$ is a tuple of probabilistic polynomial time algorithms $\mathtt{GC} = (\mathtt{GC.Garble}, \mathtt{GC.Enc}, \mathtt{GC.Eval})$ with*
- $(\Gamma, \mathtt{sk}) = \mathtt{GC.Garble}(1^\kappa, C)$ *outputs a garbled circuit Γ and a secret key* $\mathtt{sk}$.
- $c_x = \mathtt{GC.Enc}(\mathtt{sk}, x)$ *outputs an encoding c_x for an input $x \in \{0, 1\}^n$.*
- $y = \mathtt{GC.Eval}(\Gamma, c_x)$ *outputs $y = C(x)$.*

The garbling scheme $\mathtt{GC}$ *is* correct *if the probability that $\mathtt{GC.Eval}(\Gamma, c_x) \neq C(x)$ is negligible. The garbling scheme* $\mathtt{GC}$ *is* efficient *if the size of Γ is bounded by a polynomial and the run-time of $c = \mathtt{GC.Enc}(\mathtt{sk}, x)$ is also bounded by a polynomial.*

The security of garbling schemes is defined in terms of input and circuit privacy in the literature. The following definition captures the intuition that the adversary learns zero information about the circuit and input given one evaluation of the garbled circuit.

Definition 5. *(Privacy for one-time garbling schemes) A garbling scheme* $\mathtt{GC}$ *for a family of circuits $\mathcal{C}$ is said to be* input and circuit private *if there exists a probabilistic polynomial time simulator $\mathtt{Sim}_{\mathtt{Garble}}$ such that for all probabilistic polynomial time adversaries A and D and all large κ, we have*

$$\left| Pr[D(\alpha, x, C, \Upsilon, c) = 1 | \mathtt{REAL}] - Pr[D(\alpha, x, C, \tilde{\Upsilon}, \tilde{c}) = 1 | \mathtt{SIM}] \right| = \mathrm{negl}(\kappa)$$

where $\mathtt{REAL}$ *and* $\mathtt{SIM}$ *are the following events*

$$\mathtt{REAL} : (x, C, \alpha) = A(1^\kappa); (\Upsilon, \mathtt{sk}) = \mathtt{GS.Garble}(1^\kappa, C); c = \mathtt{GS.Enc}(\mathtt{sk}, x)$$
$$\mathtt{SIM} : (x, C, \alpha) = A(1^\kappa); (\tilde{\Upsilon}, \tilde{c}) = \mathtt{Sim}_{\mathtt{Garble}}(1^\kappa, C(x), 1^{|C|}, 1^{|x|}).$$

The reusable garbling schemes for circuits have the same syntax as one-time garbling schemes. In order to differentiate them, we use $\mathtt{RGC}$ to denote reusable circuit garbling schemes. The following privacy definition for reusable garbled circuits is adapted from Goldwasser et al [14].

Definition 6. *(Private reusable garbling circuits) Let* $\mathtt{RGC}$ *be a reusable garbling scheme for a family of circuits* $\mathcal{C} = \{\mathcal{C}_n\}_{n\in N}$ *and* $C \in \mathcal{C}_n$ *be a circuit with* n*-bits input. For a pair of probabilistic polynomial time algorithms* $A = (A_0, A_1)$ *and a probabilistic polynomial time simulator* $S = (S_0, S_1)$*, define two experiments:*

$\underline{\mathtt{Exp}^{\mathtt{real}}_{\mathtt{RGC},A}(1^\kappa):}$	$\underline{\mathtt{Exp}^{\mathtt{ideal}}_{\mathtt{RGC},A,S}(1^\kappa):}$
$(C, \mathtt{state}_A) \leftarrow A_0(1^\kappa)$	$(C, \mathtt{state}_A) \leftarrow A_0(1^\kappa)$
$(\mathtt{sk}, \Upsilon) \leftarrow \mathtt{RGC.Garble}(1^\kappa, C)$	$(\tilde{\Upsilon}, \mathtt{state}_S) \leftarrow S_0(1^\kappa, C)$
$\alpha \leftarrow A_1^{\mathtt{RGC.Enc(sk,\cdot)}}(M, \Upsilon, \mathtt{state}_A)$	$\alpha \leftarrow A_1^{O(\cdot,C)[[\mathtt{state}_S]]}(M, \tilde{\Upsilon}, \mathtt{state}_A)$

In the above experiments, $O(\cdot, C)[[\mathtt{state}_S]]$ *is an oracle that on input* x *from* A_1*, runs* S_1 *with inputs* $1^{|x|}$*,* $C(x)$*, and the latest state of* S*; it returns the output of* S_1 *(storing the new simulator state for the next invocation). The garbling scheme* $\mathtt{RGC}$ *is said to be* private *with reusability if there exists a probabilistic polynomial time simulator* S *such that for all pairs of probabilistic polynomial time adversaries* $A = (A_0, A_1)$*, the following two distributions are computationally indistinguishable:*

$$\left\{\mathtt{Exp}^{\mathtt{real}}_{\mathtt{RGC},A}(1^\kappa)\right\}_{\kappa\in N} =_c \left\{\mathtt{Exp}^{\mathtt{ideal}}_{\mathtt{RGC},A,S}(1^\kappa)\right\}_{\kappa\in N} \tag{1}$$

3.3 Indistinguishability obfuscation and reusable garbled circuits

Jain, Lin, and Sahai [18] proved the following results.

Theorem 1. *(Jain, Lin, and Sahai [18]) Let* τ *be arbitrary constants greater than* 0*, and* δ*,* ε *in (0, 1). Assume sub-exponential security of the following assumptions, where* κ *is the security parameter,* p *is a* κ*-bit prime, and the parameters* l, k, n *below are large enough polynomials in* κ*:*

- *the LWE assumption over* Z_p *with subexponential modulus-to-noise ratio* 2^{k^ε}*, where* k *is the dimension of the LWE secret,*
- *the LPN assumption over* Z_p *with polynomially many LPN samples and error rate* $1/l^\delta$*, where l is the dimension of the LPN secret,*
- *the existence of a Boolean* PRG *in* NC^0 *with stretch* $n^{1+\tau}$*,*
- *the SXDH assumption on asymmetric bilinear groups of a order* p*.*

Then, (subexponentially secure) indistinguishability obfuscation for all polynomial-size circuits exists.

The functional encryption scheme for circuits $C \in NC^1$ is defined as follows. Choose two standard public-key encryption key pairs $(\mathtt{puk}_1, \mathtt{prk}_1)$ and $(\mathtt{puk}_2, \mathtt{prk}_2)$ in the key generation process of the Functional Encryption scheme. The encryption of an input x consists of two ciphertexts of x under the two public keys $\mathtt{puk}_1$ and $\mathtt{puk}_2$ together with a statistically simulation sound non-interactive zero knowledge (NIZK) proof that both ciphertexts encrypt the same message. The secret key $\mathtt{sk}_C$ for the circuit C is an indistinguishability obfuscation of a program that first checks the NIZK proof and, if the proof is valid, it uses one of the two secret keys $\mathtt{prk}_1$ and $\mathtt{prk}_2$ to decrypt x and then computes and outputs $C(x)$.

A reusable garbled circuit $\overline{C}$ for a circuit $C \in NC^1$ can be constructed using the approach presented in Goldwasser et al [14]. That is, we use the techniques "from FE to reusable garbled circuits" in [14]. The owner of circuit C chooses a secret key $\mathtt{sk}$ for an ideal cipher to encrypt the circuit C as $\mathtt{E.Enc}_{\mathtt{sk}}(C)$. Let $U_E(\mathtt{sk}, x) \in NC^1$ be a universal circuit that first uses $\mathtt{sk}$ to decrypt $\mathtt{E.Enc}_{\mathtt{sk}}(C)$ to the circuit C and then runs C on x. Let $\mathtt{sk}_{U_E}$ be the secret key of the functional encryption scheme for U_E. The reusable garbled circuit $\overline{C}$ is the functioanl encryption scheme secret key $\mathtt{sk}_{U_E}$. The secret key is $(\mathtt{sk}, \mathtt{puk}_1, \mathtt{puk}_2)$. The input to $\overline{C}$ consists of the two cipher texts of $(\mathtt{sk}, x)$ under the two public keys $\mathtt{puk}_1, \mathtt{puk}_2$ and a NIZK proof that these two cipher texts encrypt the same plain text. In the above arguments, we used the fact that there exists a universal circuit of depth $O(d)$ for all circuits of depth d. By combining the results in Theorem 1, De Caro et al [8], and Goldwasser et al [14], we have the following result: With assumptions of Theorem 1, there exists a reusable garbling scheme $\mathtt{RGC}$ for circuits in NC^1 that is secure according to the Definition 6 in the random oracle model.

There exists a functional encryption scheme $\mathtt{FE}$ for circuits in NC^1 that is secure according to a standard security definition of functional encryption in the simulation-based security model (see, e.g., Katz et al [19], Bethencourt et al [3], Gorbunov et al [17], and Goldwasser et al [14]).

3.4 Construction of reusable garbled ORAMs

The syntax for reusable garbled ORAMs is the same as that for one-time and reusable garble circuits in Definition 4. The security for ORAMs is defined in Definition 1. The security definition for reusable garbled ORAMs is the same as that for reusable garble circuits in Definition 6. Throughout this paper, we will use $\mathtt{RGO} = (\mathtt{RGO.Garble}$, $\mathtt{RGO.Enc}$, $\mathtt{RGO.Eval})$ to denote a reusable garbled ORAM scheme. It is noted that a RAM CPU runs five kinds of operations. For convenience, RAM operations could be further grouped into two categories:

1. interface operation (op, v, x)
2. execute one instruction step to update the CPU state Σ and to produce the next interface operation (op, v, x). CPU state update includes register content update, program pointer update, query counter update, session information update, cryptographic key update, and other information update. The instruction step could be one of the following operations: arithmetic instruction, random sequence generation, control transfer, and halt.

Beame, Cook, and Hoover [2] showed that division could be implemented using a depth $O(\log)$ circuit. Thus the operation of one CPU step could be simulated by a circuit in NC^1.

In the ORAM model, each interface operation (op, v, x) is translated to a sequence of memory cell accesses to hide the actual data-identifier string v. Since we are trying to convert the shielded CPU to a garbled circuit, the evaluator of the garbled CPU can observe how many operations the CPU executes before the next interface command is created. In order to hide this kind of pattern, we assume that the CPU is modified in such a way that each CPU operation is followed by an interface operation. This could be achieved by inserting dummy memory cell accesses or by inserting NOP operations to the CPU instruction sequences.

Let $\mathtt{E} = (\mathtt{E.KeyGen}, \mathtt{E.Enc}, \mathtt{E.Dec})$ be a semantically secure symmetric key cipher and $\mathtt{PKE} = (\mathtt{PKE.KeyGen}, \mathtt{PKE.Enc}, \mathtt{PKE.Dec})$ be a semantically secure public key encryption scheme. Throughout the garbling process of an ORAM, we use a secret key $\mathtt{sk} = \mathtt{E.KeyGen}(1^\kappa)$ and two public key pairs

$$(\mathtt{prk}_1, \mathtt{puk}_1) = \mathtt{PKE.KeyGen}(1^\kappa)$$

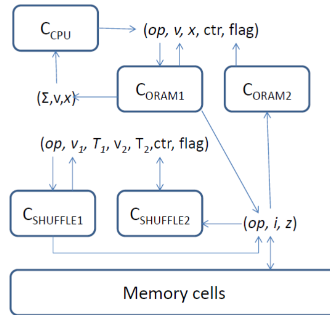
and

$$(\mathtt{prk}_2, \mathtt{puk}_2) = \mathtt{PKE.KeyGen}(1^\kappa).$$

The entire memory cells are encrypted inputs to Goldwasser et al's reusable garbled circuits (see Section 3.3 for details). That is, each cell value (v, x) is encrypted as a tuple (e_1, e_2, π) where $e_i = \mathtt{PKE.Enc}(\mathtt{puk}_i, \mathtt{sk}||v||x)$ for $i = 1, 2$ and π is a NIZK proof that both e_1 and e_2 encrypt the same message.

An ORAM CPU is modeled as three separate reusable circuits. The graphical description of these circuits and their communication channels are shown in Figure 1.

Fig. 1. Garbled ORAM CPU



The first circuit is the reusable CPU circuit $C_{\mathtt{CPU}} \in NC^1$ that takes the current CPU state (Σ, v, x) as inputs, checks the consistency of session information contained in the input, runs one CPU step, updates the CPU state Σ and session information, and produces the next encrypted interface command (op, v, x) for the second circuit $C_{\mathtt{ORAM}}$ to execute. The details of $C_{\mathtt{CPU}}$ are described in Figure 2.

The interface command (op, v, x) produced by circuit $C_{\mathtt{CPU}}$ is in the default format $(op, v, x, \mathtt{ctr}, \mathtt{flag})$ where $\mathtt{ctr} = 0$ and $\mathtt{flag} = \mathtt{no}$. The second circuit $C_{\mathtt{ORAM}}$ translates the command $(op, v, x, \mathtt{ctr}, \mathtt{flag})$ to a sequence of memory cell access commands to implement the command (op, v, x) obliviously. In order to run (op, v, x) obliviously, the circuit $C_{\mathtt{ORAM}}$ needs to keep a record on whether the actual data-identifier string v has been found. This information is kept in the $\mathtt{flag}$ field. The counter $\mathtt{ctr}$ is used to record the number of memory cells this circuit has accessed for this specific

Fig. 2. C_{CPU} updates its state and outputs an encoded $\overline{(op, v, x, \texttt{ctr}, \texttt{flag})}$

Inputs: CPU state $\underline{(\Sigma, v, x, \texttt{sctrl}_r)}$ and session control data $(\texttt{session}, \texttt{sctrl}_s)$.
Output: Encoded $\overline{(\Sigma, v, x)}$, $\overline{\texttt{session}}$, and interface command $\overline{\texttt{com}}$

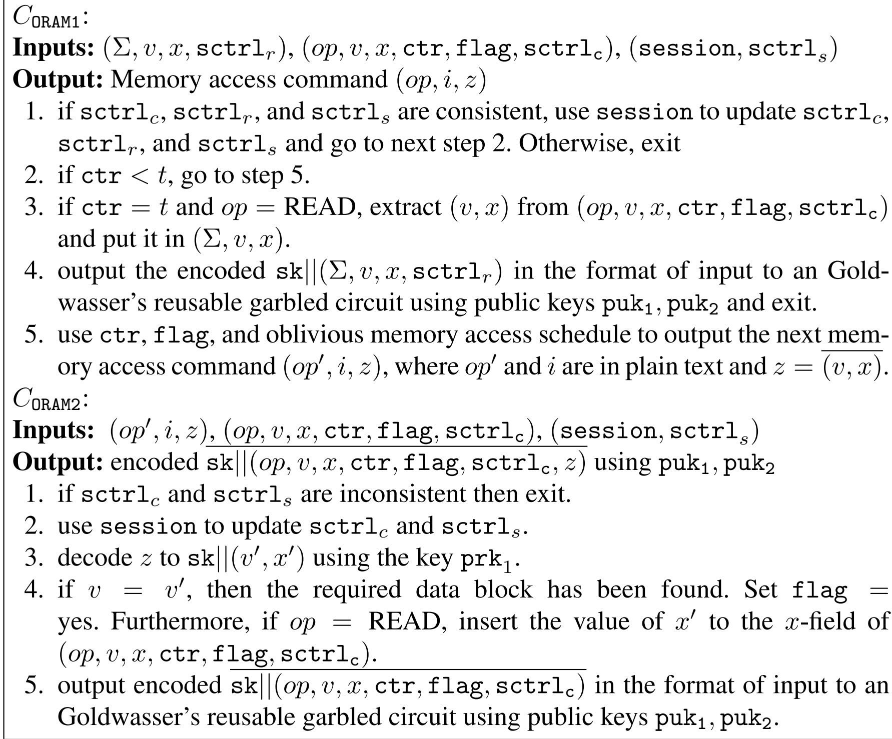
1. if $\texttt{sctrl}_r$ and $\texttt{sctrl}_s$ are inconsistent then exit.
2. simulate ORAM CPU execution for one step with internal state Σ and input (v, x), compute the new state Σ and the next interface command (op, v, x).
3. update session, $\texttt{sctrl}_r$, and $\texttt{sctrl}_s$.
4. encode $\texttt{sk}||(\Sigma, v, x)||\texttt{sctrl}_r$ to obtain $\overline{(\Sigma, v, x)} = (e_1^{\Sigma}, e_2^{\Sigma}, \pi^{\Sigma})$ as an input to a Goldwasser's reusable garbled circuit using public keys $\texttt{puk}_1, \texttt{puk}_2$.
5. encode $\texttt{sk}||\texttt{session}||\texttt{sctrl}_s$ to obtain $\overline{\texttt{session}}$ as an input to a Goldwasser's reusable garbled circuit using public keys $\texttt{puk}_1, \texttt{puk}_2$.
6. encode $\texttt{sk}||(op, v, x, \texttt{ctr}, \texttt{flag}, \texttt{strl}_{\texttt{c}})$ to obtain $\overline{\texttt{com}}$ as an input to a Goldwasser's reusable garbled circuit using public keys $\texttt{puk}_1, \texttt{puk}_2$.
7. output $\overline{(\Sigma, v, x)}$, $\overline{\texttt{session}}$, and $\overline{\texttt{com}}$.

command (op, v, x). In other words, for each (op, v, x) command, the circuit $C_{\texttt{ORAM}}$ is executed t times repeatedly to hide the actual data block it accessed, where t is a constant that is independently of the value (op, v, x).

Specifically, the reusable circuit $C_{\texttt{ORAM}}$ is a pair of circuits $C_{\texttt{ORAM1}}$ and $C_{\texttt{ORAM2}}$. $C_{\texttt{ORAM1}}$ takes the interface command $(op, v, x, \texttt{ctr}, \texttt{flag})$ as the input and produces an interface command (op', i, z) where op' and i are plain texts and $z = \overline{(v, x)}$ is encoded. If op' =READ, the evaluator reads the memory cell at physical location i and loads the encoded content to z. If op' =WRITE, the evaluator writes the encoded value z to the memory cell at physical location i. After the evaluator finishes processing the actual memory cell access, the resulting updated interface command (op', i, z) is given to the circuit $C_{\texttt{ORAM2}}$. $C_{\texttt{ORAM2}}$ decrypts z to a pair (v', x'). If $v = v'$, the actual data block has been found. $C_{\texttt{ORAM2}}$ sets $\texttt{flag} =$ yes and checks whether $op =$ READ. If $op =$ READ, $C_{\texttt{ORAM2}}$ needs to copy the value x' to the field x of $(op, v, x, \texttt{ctr}, \texttt{flag})$. If $v \neq v'$, the actual data block has not been found yet and $C_{\texttt{ORAM2}}$ keeps $\texttt{flag} =$ no. After $C_{\texttt{ORAM2}}$ finishes its job, circuit $C_{\texttt{ORAM1}}$ takes turn again. $C_{\texttt{ORAM1}}$ examines the interface command $(op, v, x, \texttt{ctr}, \texttt{flag})$ to obtain the values of flag and ctr. Using these values, $C_{\texttt{ORAM1}}$ creates the next memory access instruction according to the oblivious memory access schedule and outputs the corresponding interface command (op', i, z). For each execution of $C_{\texttt{ORAM1}}$, the counter ctr is increased by one. After ctr reaches t, $C_{\texttt{ORAM1}}$ copies the value x from $(op, v, x, \texttt{ctr}, \texttt{flag})$ to the field-x of (Σ, v, x) which is the input to $C_{\texttt{CPU}}$. The evaluator knows the value of t. Thus it lets the circuit $C_{\texttt{CPU}}$ take turn after $C_{\texttt{ORAM1}}$ finishes t steps. The details of circuit $C_{\texttt{ORAM}}$ are described in Figure 3.

The third circuit $C_{\texttt{SHUFFLE}}$ implements the re-shuffling process for ORAM memory cells. $C_{\texttt{SHUFFLE}}$ consists of a pair $(C_{\texttt{SHUFFLE1}}, C_{\texttt{SHUFFLE2}})$ of circuits that implement the randomized data-oblivious Shellsort algorithm for re-shuffling the memory cells. $C_{\texttt{SHUFFLE}}$ is constructed using an oblivious sorting algorithm. Specifically, $C_{\texttt{SHUFFLE1}}$ takes an input with fields $(op, v_1, T_1, v_2, T_2, \texttt{ctr}, \texttt{flag})$ and compare-exchanges the

Fig. 3. $C_{\texttt{ORAM}}$ uses ctr and flag to determine which memory cell to access



$C_{\texttt{ORAM1}}$:

Inputs: $(\Sigma, v, x, \texttt{sctrl}_r)$, $(op, v, x, \texttt{ctr}, \texttt{flag}, \texttt{sctrl}_{\texttt{c}})$, $(\texttt{session}, \texttt{sctrl}_s)$

Output: Memory access command (op, i, z)

1. if $\texttt{sctrl}_c$, $\texttt{sctrl}_r$, and $\texttt{sctrl}_s$ are consistent, use session to update $\texttt{sctrl}_c$, $\texttt{sctrl}_r$, and $\texttt{sctrl}_s$ and go to next step 2. Otherwise, exit
2. if $\texttt{ctr} < t$, go to step 5.
3. if $\texttt{ctr} = t$ and $op =$ READ, extract (v, x) from $(op, v, x, \texttt{ctr}, \texttt{flag}, \texttt{sctrl}_{\texttt{c}})$ and put it in (Σ, v, x).
4. output the encoded $\texttt{sk}||(\Sigma, v, x, \texttt{sctrl}_r)$ in the format of input to an Goldwasser's reusable garbled circuit using public keys $\texttt{puk}_1, \texttt{puk}_2$ and exit.
5. use ctr, flag, and oblivious memory access schedule to output the next memory access command (op', i, z), where op' and i are in plain text and $z = \overline{(v, x)}$.

$C_{\texttt{ORAM2}}$:

Inputs: (op', i, z), $(op, v, x, \texttt{ctr}, \texttt{flag}, \texttt{sctrl}_{\texttt{c}})$, $(\texttt{session}, \texttt{sctrl}_s)$

Output: encoded $\overline{\texttt{sk}||(op, v, x, \texttt{ctr}, \texttt{flag}, \texttt{sctrl}_{\texttt{c}}, z)}$ using $\texttt{puk}_1, \texttt{puk}_2$

1. if $\texttt{sctrl}_c$ and $\texttt{sctrl}_s$ are inconsistent then exit.
2. use session to update $\texttt{sctrl}_c$ and $\texttt{sctrl}_s$.
3. decode z to $\texttt{sk}||(v', x')$ using the key $\texttt{prk}_1$.
4. if $v = v'$, then the required data block has been found. Set $\texttt{flag} =$ yes. Furthermore, if $op =$ READ, insert the value of x' to the x-field of $(op, v, x, \texttt{ctr}, \texttt{flag}, \texttt{sctrl}_{\texttt{c}})$.
5. output encoded $\overline{\texttt{sk}||(op, v, x, \texttt{ctr}, \texttt{flag}, \texttt{sctrl}_{\texttt{c}})}$ in the format of input to an Goldwasser's reusable garbled circuit using public keys $\texttt{puk}_1, \texttt{puk}_2$.

values (v_1, T_1) and (v_2, T_2). After the compare-exchange operation, $C_{\texttt{SHUFFLE1}}$ outputs a next memory cell access command (op, i, z) with plain-text op and i for the evaluator to access the ith memory cell. The updated (op, i, z) is given to $C_{\texttt{SHUFFLE2}}$ that decodes z and inserts it to the corresponding field of $(op, v_1, T_1, v_2, T_2, \texttt{ctr}, \texttt{flag})$ if necessary. The algorithm for $C_{\texttt{SHUFFLE}}$ is similar to the algorithm for the circuit $C_{\texttt{ORAM}}$ described in Figure 3. The details are omitted here.

Without loss of generality, we may assume that there exist NC^1 encryption and decryption circuits for the symmetric key cipher E and the public key cipher PKE. Then it is straightforward to show that circuits $C_{\texttt{CPU}}$, $C_{\texttt{ORAM1}}$, $C_{\texttt{ORAM2}}$, $C_{\texttt{SHUFFLE1}}$, and $C_{\texttt{SHUFFLE2}}$ belong to NC^1 also. By Goldwasser et al's results that we discussed in Section 3.3, there exist efficient polynomial size reusable garbled circuits $\bar{C}_{\texttt{CPU}}$, $\bar{C}_{\texttt{ORAM1}}$, $\bar{C}_{\texttt{ORAM2}}$, $\bar{C}_{\texttt{SHUFFLE1}}$, and $\bar{C}_{\texttt{SHUFFLE2}}$. Using these constructions, we are ready to give the formal construction of our reusable garbled ORAM machines.

In above paragraphs, we described the construction of a garbled ORAM CPU. In a practical deployment of ORAM programs, the program is generally encoded and stored in the memory cells. In other words, the garbled ORAM CPU could be considered as a garbled universal machine. It reads encoded ORAM programs in memory cells and

executes them on inputs. Part of the input is provided by the client and the other part of the input is located in memory cells already. For example, part of memory cells may be considered as an encoded database which is part of the input to the ORAM program. The other part of the input could be an encoded database search query that is submitted by the client. Let $\mathcal{P} = \{\mathcal{P}_n\}_{n\in N}$ be a family of ORAM programs such that P_n is a set of programs that take n-bit inputs. The reusable garbling scheme $\texttt{RGO} = (\texttt{RGO.Garble}, \texttt{RGO.Enc}, \texttt{RGO.Eval})$ for $\mathcal{P}$ is instantiated as follows.

- $(\Gamma, (\texttt{sk}, \texttt{puk}_1, \texttt{puk}_2)) = \texttt{RGO.Garble}(1^\kappa, P)$:
  - $\texttt{sk} = \texttt{E.KeyGen}(1^\kappa)$
  - $(\texttt{prk}_i, \texttt{puk}_i) = \texttt{PKE.KeyGen}(1^\kappa)$ for $i = 1, 2$
  - encode P appropriately and include it as part of the memory cells
  - encode each memory cell content (v, x) to (e_1, e_2, π) which is in the format of input to Goldwasser et al's reusable garbled circuits using $\texttt{sk}, \texttt{puk}_1$, and $\texttt{puk}_2$.
  - use Goldwasser et al's approach to construct reusable garbled circuits for each of the NC^1 circuits $C_{\texttt{CPU}}$, $C_{\texttt{ORAM1}}$, $C_{\texttt{ORAM2}}$, $C_{\texttt{SHUFFLE1}}$, $C_{\texttt{SHUFFLE2}}$ with keys $\texttt{prk}_1$, $\texttt{sk}$, $\texttt{puk}_1$, and $\texttt{puk}_2$.
  - Let $\Gamma = \left(\bar{C}_{\texttt{CPU}}, \bar{C}_{\texttt{ORAM1}}, \bar{C}_{\texttt{ORAM2}}, \bar{C}_{\texttt{SHUFFLE1}}, \bar{C}_{\texttt{SHUFFLE2}}\right)$.
- $c = \texttt{RGO.Enc}((\texttt{sk}, \texttt{puk}_1, \texttt{puk}_2), x)$.
  - Encode $\texttt{sk}||x$ to $c = (e_1, e_2, \pi)$ which is in the format of input to Goldwasser et al's reusable garbled circuits using the two keys $\texttt{puk}_1$, and $\texttt{puk}_2$.
- $y = \texttt{RGO.Eval}(\Gamma, c)$:
  - run the garbled ORAM CPU

$$\left(\bar{C}_{\texttt{CPU}}, \bar{C}_{\texttt{ORAM1}}, \bar{C}_{\texttt{ORAM2}}, \bar{C}_{\texttt{SHUFFLE1}}, \bar{C}_{\texttt{SHUFFLE2}}\right)$$

    on the memory cells and on c to compute the output $y = P(x)$.

3.5 Proof of security

We first make a few observations on the garbled ORAM construction in Section 3.4. The first observation is that for a given encoded input $c = \texttt{RGO.Enc}(x)$, the running time of $y = \texttt{RGO.Eval}(\Gamma, c)$ is disclosed according to Definition 1. Thus the running time is provided to the simulator in advance. In case that the running time of the execution should be protected also, the ORAM CPU should be revised in such a way that it takes the same time for all inputs of the same length. This could be achieved by adding NOP operations to the ORAM CPU if the calculation ends early than the expected time.

The second observation is that the output $y = \texttt{RGO.Eval}(\Gamma, c)$ is an encoded value (e_1, e_2, π) which is in the format of input to Goldwasser et al's reusable garbled circuits using the keys $\texttt{sk}, \texttt{puk}_1$, and $\texttt{puk}_2$. This is acceptable in practice since generally the garbled ORAM program is executed in the cloud and the cloud does not need to know the actual output. After the computation is finished, the cloud returns the encoded y to the client who can recover the plain text output using either of the secret keys $\texttt{prk}_1$ or $\texttt{prk}_2$.

The third observation is that in our scheme, the garbled ORAM will only run on encoded input provided by the client. The secret key $\texttt{sk}$ of the ideal cipher $\texttt{E}$ and the public keys $\texttt{puk}_1, \texttt{puk}_2$ of the public key scheme $\texttt{PKE}$ are needed to encode the input

for the garbled ORAM CPUs. Without correctly encoded inputs with matching session identification, neither garbled CPU circuit $\overline{C}_{\texttt{CPU}}$ nor garbled ORAM circuit $\overline{C}_{\texttt{ORAM}}$ would continue the computation since the session control message validation process would only pass with a negligible probability. Similarly, the adversary could not mix/swap computation states for two inputs since each input contains an input specific session control message. The session control messages in inputs for two sessions (even if the input values are identical) are identical only with a negligible probability.

The fourth observation is that the adversary may play fault-insertion attacks in the memory cells. This kind of attacks have not been discovered or modeled in the traditional simulation-based security model for ORAMs in Definitions 1 and 6. In this section, we prove that our ORAM garbling scheme is secure according to Definition 6.

Theorem 2. *Assuming the existence of a semantically secure symmetric key cipher* E*, a semantically secure public key cipher* PKE*, and a private reusable garbling scheme for circuits in NC^1, there is a private reusable ORAM garbling scheme* RGO *as defined in Definition 6 in the random oracle model.*

Proof. First we observe that the existence of semantically secure ciphers E and PKE implies the existence of cryptographically secure one-way functions. Thus the assumption in Theorem 2 implies the existence of a secure ORAM according to Definitions 1. The correctness of the construction in Section 3.4 is straightforward. In order to show that the construction is input and circuit private as defined in Definition 6, we show that there exists a simulator $S = (S_0, S_1)$ simulating the garbled execution given the program output y and the ORAM CPU running time t, so that the equation (1) holds.

Let S_a be the ORAM memory access pattern simulator and $S_{\texttt{PKE}}$ be the simulator for the cipher PKE. Let $S_{\texttt{CPU}}, S_{\texttt{ORAM}}$, and $S_{\texttt{SHUFFLE}}$ be the simulators for Goldwasser et al's reusable garble circuits (as described in Section 3.3) $\overline{C}_{\texttt{CPU}}, \overline{C}_{\texttt{ORAM}}$, and $\overline{C}_{\texttt{SHUFFLE}}$ respectively.

Assume that an ORAM machine P is selected with the security parameter κ. For the given ORAM CPU running time t and output $y = P(x)$, let $S_a(y, t)$ outputs a sequence of memory access pattern $\eta_1, \cdots, \eta_t$ where η_i $(i \leq t)$ is the simulated oblivious memory cell access sequence for the ith memory cell access of the original RAM machine. In other words, each $\eta_i = \{v_{i,1}, \cdots, v_{i,t_i}\}$ is a sequence of memory cells that the simulated ORAM machine accesses to implement the ith memory cell access of the original RAM machine. $S_a(y, t)$ also outputs a sequence of memory access pattern $\xi_1, \cdots, \xi_{t'}$ where ξ_i $(i \leq t')$ is the simulated oblivious memory cell access sequence for the ith re-shuffling process.

Starting from the last memory cell sequence set η_t, for each $\eta_i = \{v_{i,1}, \cdots, v_{i,t_i}\}$, repeat simulators $S_{\texttt{PKE}}$ and $S_{\texttt{ORAM}}$ for t_i times to generate a simulated $\texttt{view}^i_{\texttt{ORAM}}$. Similarly, starting from the last memory cell sequence set $\xi_{t'}$, for each $\xi_i = \{u_{i,1}, \cdots, u_{i,s_i}\}$, repeat simulators $S_{\texttt{PKE}}$ and $S_{\texttt{SHUFFLE}}$ for s_i times to generate a simulated $\texttt{view}^i_{\texttt{SHUFFLE}}$. Lastly, using the views $\{\texttt{view}^i_{\texttt{ORAM}}, \texttt{view}^j_{\texttt{SHUFFLE}} : i \leq t \text{ and } j \leq t'\}$, repeat simulators $S_{\texttt{PKE}}$ and $S_{\texttt{CPU}}$ for $t + t'$ times to generate a simulated $\texttt{view}_{\texttt{CPU}}$. Without loss of generality, we may assume that the adversaries output their entire views in the above simulation so that any required view could be calculated from these views in a probabilistic polynomial time. In other words, the simulator's view $\left\{\texttt{Exp}^{\texttt{ideal}}_{\texttt{RGO},A,S}(1^\kappa)\right\}_{\kappa \in N}$ in (1) could

be calculated in probabilistic polynomial time from S's entire view

$$\{\mathtt{view}^i_{\mathtt{ORAM}}, \mathtt{view}^j_{\mathtt{SHUFFLE}}, \mathtt{view}_{\mathtt{CPU}} : i \le t \text{ and } j \le t'\}_{\kappa \in N}. \quad (2)$$

In order to show that (1) holds, we consider three experiments.
Experiment 1: The ideal game $\mathtt{Exp}^{\mathtt{ideal}}_{\mathtt{RGO},A,S}(1^\kappa)$ of Definition 6 with the simulator S and the ORAM machine P.
Experiment 2: The same as Experiment 1 except that the ORAM program P is replaced with the reusable garbled ORAM $\overline{P}$: $\overline{C}_{\mathtt{CPU}}, \overline{C}_{\mathtt{ORAM}}, \overline{C}_{\mathtt{SHUFFLE}}$, and corresponding keys.
Experiment 3: The same as Experiment 2 except that the simulated cipher $S_{\mathtt{PKE}}$ is replaced with the actual cipher PKE using keys $\mathtt{sk}, \mathtt{puk}_1, \mathtt{puk}_2, \mathtt{prk}_1$.

Since the view of Experiment 1 equals to $\left\{\mathtt{Exp}^{\mathtt{ideal}}_{\mathtt{RGO},A,S}(1^\kappa)\right\}_{\kappa \in N}$ and the view of Experiment 3 equals to $\left\{\mathtt{Exp}^{\mathtt{real}}_{\mathtt{RGO},A}(1^\kappa)\right\}_{\kappa \in N}$, it is sufficient for us to show that the view of Experiment 1 is computationally indistinguishable from the view of Experiment 2 and the view of Experiment 2 is computationally indistinguishable from the view of Experiment 3.

Claim. Assume that $\overline{C}_{\mathtt{CPU}}, \overline{C}_{\mathtt{ORAM}}$, and $\overline{C}_{\mathtt{SHUFFLE}}$ are private reusable garbled circuits for circuits $C_{\mathtt{CPU}}, C_{\mathtt{ORAM}}$, and $C_{\mathtt{SHUFFLE}}$. Furthermore, assume that the ORAM access pattern is securely simulated by the simulator S_a. Then the outputs of Experiment 1 and Experiment 2 are computationally indistinguishable.

Proof outline. Assume that outputs of Experiment 1 and Experiment 2 could be distinguished by a probabilistic polynomial time algorithm D. Then a standard hybrid approach could be used to construct a probabilistic polynomial time algorithm D' to distinguish the view in (2) from the view for the ideal experiments with the ORAM program P. In other words, if S_a securely simulate the ORAM memory cell access pattern, then the view in (2) could be distinguished from the ideal experiments with circuits $C_{\mathtt{CPU}}, C_{\mathtt{ORAM}}$, and $C_{\mathtt{SHUFFLE}}$. This contradicts the fact that $\overline{C}_{\mathtt{CPU}}, \overline{C}_{\mathtt{ORAM}}$, and $\overline{C}_{\mathtt{SHUFFLE}}$ are private reusable garbled circuits (see Section 3.3). Q.E.D.

Claim. Assume that the cipher PKE be semantically secure. Then the outputs of Experiment 2 and Experiment 3 are computationally indistinguishable.

Proof outline. Assume that there exist probabilistic polynomial time adversaries $A = (A_1, A_2)$ and a probabilistic polynomial time distinguisher D such that D can distinguish the outputs of Experiment 2 and Experiment 3 with a non-negligible probability. Using the standard hybrid argument, one can construct a probabilistic polynomial time distinguisher D' to distinguish at least one cipher text in Experiment 3 from the corresponding simulated cipher text in Experiment 2 with a non-negligible probability. This contradicts the assumption that PKE is semantically secure. Q.E.D.

Claim 1 and Claim 2 imply that the equation (1) holds. This completes the proof of Theorem 2. Q.E.D.

Definition 7. *(Private reusable garbling ORAMs) Let* RGO *be a reusable garbling scheme for ORAM machines. In addition to the attacks that are allowed in Definitions 1 and 6, the adversary is allowed to interfere with the garbled ORAM execution by playing or*

replaying the garbled ORAM on modified environments (e.g., inserting faults in the memory cells during the execution). The garbling scheme $\mathtt{RGO}$ *is said to be* private *with reusability if there exists a probabilistic polynomial time simulator S such that for all pairs of probabilistic polynomial time adversaries $A = (A_0, A_1)$ as defined above, the two distributions in (1) are computationally indistinguishable.*

Theorem 3. *Assume the existence of a semantically secure symmetric key cipher* $\mathtt{E}$*, a semantically secure public key cipher* $\mathtt{PKE}$*, a secure digital signature scheme* $\mathtt{Dsig}$*, and a private reusable garbling scheme for circuits in NC^1. Then there is a private reusable ORAM garbling scheme* $\mathtt{RGO}_1$ *as defined in Definition 7 in the random oracle model.*

Proof. Let $\mathtt{Dsig} = (\mathtt{Dsig.KeyGen}, \mathtt{Dsig.Sign}, \mathtt{Dsig.Vefy})$ be a secure digital signature scheme and $(\mathtt{SIGsk}, \mathtt{SIGpk}) = \mathtt{Dsig.KeyGen}(1^\kappa)$. Let $\mathcal{P} = \{\mathcal{P}_n\}_{n\in N}$ be a family of square-root ORAM programs such that P_n is a set of functions that take n-bit inputs. The reusable garbling scheme $\mathtt{RGO}_1 = (\mathtt{RGO}_1.\mathtt{Garble}, \mathtt{RGO}_1.\mathtt{Enc}, \mathtt{RGO}_1.\mathtt{Eval})$ for $\mathcal{P}$ is instantiated as in Section 3.4 with the following revisions:

- $(\Gamma, (\mathtt{sk}, \mathtt{puk}_1, \mathtt{puk}_2, \mathtt{SIGsk}, \mathtt{SIGpk})) = \mathtt{RGO}_1.\mathtt{Garble}(1^\kappa, P)$: This process is obtained from $\mathtt{RGO.Garble}$ in Section 3.4 by the following revisions:
  - add a component in circuit $C_{\mathtt{ORAM}}$ to digitally sign the shelter (cache) at the end of the execution of each $C_{\mathtt{CPU}}$ interface command (op, v, x)
  - add a component in circuit $C_{\mathtt{ORAM}}$ to check the validity of the digital signature at the beginning of the execution of each $C_{\mathtt{CPU}}$ interface command (op, v, x).
  - add an component to circuit $C_{\mathtt{SHUFFLE}}$ so that a unique sequence number $\mathtt{seq}$ is added to all memory cells. At the same time, the physical location i of the memory cell $OR[i]$ is added to the content of $OR[i]$. In other words, the ith memory cell contains the value $OR[i] = (v, x, \mathtt{seq}, i)$.
  - add a component to circuit $C_{\mathtt{ORAM}}$ to check that the accessed non-shelter memory cells contain the current sequence number and check that the actual physical address of the memory cell is the same as that contained in the content $OR[i] = (v, x, \mathtt{seq}, i)$.
- $c = \mathtt{RGO}_1.\mathtt{Enc}((\mathtt{sk}, \mathtt{puk}_1, \mathtt{puk}_2), x)$. This is obtained from $\mathtt{RGO.Enc}$ by adding a process to digitally sign the entire shelter cells and adding the current sequence number to all non-shelter memory cells if this has not been done yet.
- $y = \mathtt{RGO}_1.\mathtt{Eval}(\Gamma, c)$: same as $\mathtt{RGO.Eval}(\Gamma, c)$.

The remaining part of the proof is similar to the proof of Theorem 2. Q.E.D.

4 Witness encryption and extractable witness encryption

Witness encryption schemes are based on languages in $\mathtt{NP}$. For each language $L \in \mathtt{NP}$, there is a polynomial witness relation R. That is, $x \in L$ if and only if there is a w such that $(x, w) \in R$. In a witness scheme, one can encrypt a message M with a string x to get a cipher text $c = E(M, x, L)$. The recipient of c can decrypt M if $x \in L$ and the recipient knows a witness w with $(x, w) \in R$. Note that the encrypted does not know whether $x \in L$.

Garg et al [9] designed two witness encryption schemes based on the hardness of decision multilinear no-exact-cover problems: one is based on multilinear group families and the other is based on graded encoding systems. In the following, we briefly/informally discuss the witness encryption scheme based on multilinear group families. For a given security parameter κ, a multilinear group family is a list of groups $G_1, \cdots, G_n$ of prime order $p = p(\kappa)$ with generators $g_1, g_2, \cdots, g_n$, and a multilinear map e with the properties

$$e(g_i^a, g_j^b) = g_{i+j}^{ab} \text{ for all } a, b \in Z_p.$$

The NP-complete problem Exact Cover is defined as follows: for a given collection X of subsets of the set $\{1, \cdots, n\}$, find a sub-collection $X^* \subseteq X$ such that X^* is a partition of $\{1, \cdots, n\}$.
$\texttt{Enc}_{\texttt{WE}}(1^\kappa, X, m)$: Given an Exact Cover instance $X = \{X_1, \cdots, X_l\}$ and a message m, generate a multilinear group family $G_1, \cdots, G_n$ of prime order $p = p(\kappa)$ with generators $g_1, g_2, \cdots, g_n$, and a multilinear map e. Chooses random $a_1, \cdots, a_n \in Z_p$. Assume that $m \in G_n$, the cipher text ct consists of a description of the multilinear group family, the description of X, and

$$c = m \cdot g_n^{a_1 \cdots a_n} \text{ and for all } i \in \{1, \cdots, l\}, c_i = \left(g_{|X_i|}\right)^{\prod_{j \in X_i} a_i}.$$

$\texttt{Dec}_{\texttt{WE}}(ct, X^*)$: Given the cipher text ct and the witness $X^* = \{j_1, \cdots, j_{|X^*|}\}$ associated with the partition of $\{1, \cdots, n\}$, it outputs

$$m = c/e(c_{j_1}, \cdots, c_{j_{|X^*|}}).$$

Using the witness encryption scheme, Garg, et al [9] constructed an Attribute-Based Encryption scheme for any circuits and constructed other cryptographic schemes such as public key encryption schemes, identity based encryption schemes, and fully secure identity-based encryption schemes.

Goldwasser et al [13] extended Garg et al's witness encryption scheme to extractable witness encryption by requiring that if one can recover M from $\texttt{Enc}_{\texttt{WE}}(1^\kappa, X, M)$, then one can extract a witness $X^* = \{j_1, \cdots, j_{|X^*|}\}$ for X.

5 ABE for Turing machines and RAMs

Using witness encryption schemes, Goldwasser et al [13] designed succinct Attribute Based Encryption scheme for any Turing machines and RAM machines. It is convenient to first describe the non-succinct ABE scheme first. The master secret and public keys for the ABE scheme is the pair $(\texttt{sigK}, \texttt{SigPubK})$ for a digital signature scheme. The secret function key for a Turing machine M is the digital signature σ on M under the key $\texttt{sigK}$. In order for a sender to encrypt a message m with the public attribute x, the sender computes the witness encryption $c = \texttt{Enc}_{\texttt{WE}}(1^\kappa, x^*, m)$ where $x^* = (x, \texttt{SigPubK})$. A valid witness for x^* is a tuple (M, σ, π) where M is a Turing machine, σ is the digital signature σ on M, and π is a tableau of computation proving that $M(x) = 1$. This scheme is not succinct since it takes a long time to check that π is a tableau of computation for $M(x) = 1$. Thus the size of the ciphertext c depends on the worst-case running time of M.

In order to address the succinctness challenge, Goldwasser et al used SNARK scheme in Bitansky et al [?] where SNARK (Succinct Non-interactive Arguments of Knowledge) has a securely generated common reference string $\texttt{crs}$ such that any one can prove a NP statement by presenting a proof π. The length of $\texttt{crs}$, the length of π, and the time to check the proof π only depends on the security parameter κ instead of the running time to check the NP witness. Furthermore, if an adversary can prove that x is the member of a NP language L, then the adversary can extract a witness for x. Now the succinct ABE scheme for both uniform and non-uniform Turing machines are as follows: the cipher text $c = \texttt{Enc}_{\texttt{WE}}(1^\kappa, x^*, m)$ where $x^* = (x, \texttt{crs}, \texttt{SigPubK})$. A valid witness for x^* is a tuple (π, t) where π is a proof-of-knowledge of a Turing machine M and a signature σ such that σ is the digital signature on M and M halts on input x with output 1 in at most t steps. The witness size and verification time is independent of the Turing machine size or its run time. Thus it is a succinct ABE scheme. The ABE scheme for RAM machines could be similarly defined.

6 Conclusion

In this paper, we designed reusable garbling schemes for ORAMs using alternative techniques. The garbled ORAM design could find a variety of applications in secure cloud computing environments.

References

1. M. Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In *Proc. 42nd ACM STOC*, pages 181–190. ACM, 2010.
2. P.W. Beame, S. A Cook, and H.J. Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15(4):994–1003, 1986.
3. J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *IEEE Security and Privacy, 2007.*, pages 321–334. IEEE, 2007.
4. D. Boneh, D. Mazieres, and R.A. Popa. Remote oblivious storage: Making oblivious RAM practical. In *http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf*, 2011.
5. B. Chor and N. Gilboa. Computationally private information retrieval. In *Proc. 29th STOC*, pages 304–313. ACM, 1997.
6. B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *JACM*, 45(6):965–981, 1998.
7. I. Damgård, S. Meldgaard, and J.B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography*, pages 144–163. Springer, 2011.
8. A. De Caro, V. Iovino, A. Jain, A. O'Neill, O. Paneth, and G. Persiano. On the achievability of simulation-based security for functional encryption. In *CRYPTO 2013*, pages 519–535. Springer, 2013.
9. S. Garg, S. C. Gentry, A. Sahai, and B. Waters. Witness encryption and its applications. In *Proc. 45th ACM STOC*, pages 467–476. Springer, 2013.
10. C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled RAM revisited. In *EUROCRYPT 2014*, pages 405–422. Springer, 2014.
11. O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proc. 19th ACM STOC*, pages 182–194. ACM, 1987.

12. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *JACM*, 43(3):431–473, 1996.
13. S. Goldwasser, Y. Kalai, R. Popa, V. Vaikuntanathan, and N. Zeldovich. How to run Turing machines on encrypted data. In *Proc. CRYPTO*, pages 536–553. 2013.
14. S. Goldwasser, Y. Kalai, R. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proc. 45th STOC*, pages 555–564. ACM, 2013.
15. M.T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *Proc. 21st ACM-SIAM SODA*, pages 1262–1277. SIAM, 2010.
16. M.T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Proc. ICALP*, pages 576–587. 2011.
17. S. Gorbunov, V. Vaikuntanathan, and H. Wee. Functional encryption with bounded collusions via multi-party computation. In *Proc CRYPTO*, pages 162–179. Springer, 2012.
18. A. Jain, H. Lin, and A. Sahai. Indistinguishability obfuscation from well-founded assumptions. In *Proc. 53rd Annual ACM STOC*, pages 60–73, 2021.
19. J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *EUROCRYPT 2008*, pages 146–162. Springer, 2008.
20. S.T. Kent. Protecting externally supplied software in small computers. Technical report, DTIC Document, 1980.
21. A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, 2009.
22. E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *focs*, volume 97, pages 364–373, 1997.
23. T. Leighton and C.G. Plaxton. Hypercubic sorting networks. *SIAM J. Comput.*, 27(1):1–47, 1998.
24. S. Lu and R. Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, volume 7881, pages 719–734. Springer, 2013.
25. R. Ostrovsky. *Software protection and simulation on oblivious RAMs*. PhD thesis, 1992.
26. R. Ostrovsky and V. Shoup. Private information storage. In *Proc. 29th ACM STOC*, pages 294–303. ACM, 1997.
27. R. Ostrovsky and W.E Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *PKC*, pages 393–411. Springer, 2007.
28. R. Pagh and F.F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.
29. B. Pinkas and T. Reinman. Oblivious RAM revisited. In *Proc. CRYPTO*, pages 502–519. Springer, 2010.
30. N. Pippenger and M. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.
31. V.R. Pratt. Shellsort and sorting networks. *PhD thesis, Stanford University*, 1972.
32. D.L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, 1959.