How to Garble RAM Programs*

Steve Lu[†] Rafail Ostrovsky[‡]

Abstract

Assuming solely the existence of one-way functions, we show how to construct Garbled RAM Programs (GRAM) where its size only depends on fixed polynomial in the security parameter times the program running time. We stress that we avoid converting the RAM programs into circuits. As an example, our techniques implies the first garbled *binary search* program (searching over sorted encrypted data stored in a cloud) which is poly-logarithmic in the data size instead of linear. Our result requires the existence of one-way function and enjoys the same non-interactive properties as Yao's original garbled circuits.

Keywords: Secure Computation, Oblivious RAM, Garbled Circuits.

*Patent Pending

[†]Stealth Software Technologies, Inc. E-mail: steve@stealthsoftwareinc.com

[‡]Department of Computer Science and Department of Mathematics, UCLA. Work done while consulting for Stealth Software Technologies, Inc. E-mail: rafail@cs.ucla.edu

1 Introduction

Often times, such as in cloud computation, one party wants to store some data remotely and then have the remote server perform computations on that data. If the client does not wish to reveal this data or the nature of the computation and the results of the computation to the remote server, then one must resort to using secure computation methods in order to process this remotely stored data. In other words, suppose two parties want to compute some program π on their private inputs without revealing to each other (or just one party) anything but the output. The earliest research in secure two-party computation modeled π as a circuit and was accomplished under Yao's Garbled Circuits [40] or the Goldreich-Micali-Wigderson [14] paradigm. Both of these approaches require the program π to be converted to a circuit. Even the recent work of performing secure computation via fully homomorphic encryption requires representing the program π as a circuit. However, many algorithms are more naturally and compactly represented as RAM programs, and converting these into circuits may lead to a huge blowup in program size and its running time.

Of course, there are known polynomial transformations between time-bounded RAM programs, time-bounded Turing Machines and circuits [11, 32]: Given a T-time RAM program, [11] shows how one can transform it into a $O(T^3)$-time TM, and [32] shows how to transform a T-time TM into circuits of size $O(T \log T)$, which results in a $O(T^3 \log T)$ blowup. Our work aims at *circumventing* these transformation costs and executing RAM programs directly in a private manner, while retaining the same noninteractive properties as Yao's Garbled circuits. This goal is especially important for the case of complex real-world RAM programs with running time that is much larger than the input size. Unrolling these complicated RAM programs with multiple execution paths, recursion, multiple loops, etc. into a circuit makes the circuit size polynomially larger and often prohibitive.

It should be noted that our work is also important in practical applications where the sizes of the inputs are vastly different, such as database search, or where multiple queries against the same large data-set must be executed. When compiling a RAM program into a circuit, the compiled circuit must inherently be able to compute all execution paths of the RAM program. Thus, the circuit itself must be at least be as large as the input size, which in some applications may be is exponentially larger than execution path of the insecure solution (e.g. consider a binary search). One can argue that even if the circuit is large, we can "charge" the large circuit cost to the large input size, but in many cases this is unacceptable: consider the case where a large data is encrypted and uploaded *off-line*, such as a large database, and multiple encrypted queries are made *on-line*, where the insecure execution path is, for example, poly-logarithmic in the database size and we do not want to "pay" an on-line cost of circuit size which is linear in the database size.

An alternative approach for secure conversion of RAM programs into circuits is dynamic evaluation: even if the resulting circuit is large and the total size of the is resulting circuit is prohibitive, one can execute and even compile the large circuit dynamically and intelligently evaluate only parts of the circuit so as to "prune off" dead paths (e.g. short-circuiting techniques) to make the evaluation efficient, even in the case of large inputs. However, until now it was not known how to convert RAM programs into circuits which result in an efficient secure non-interactive execution in a way that does not reveal the execution path of the compiled RAM program. Naturally, using interaction, one can use the Goldreich-Micali-Wigderson [14] paradigm along with revealing bits along the way to help prune and determine execution path – however our ultimate goal is to explore the non-interactive garbling solutions for RAM programs without revealing the execution path.

Another alternative method for computing RAM programs without first converting them to circuits was proposed by Ostrovsky and Shoup [30] which used Oblivious RAM [15] as a building block. The Ostrovsky-Shoup compiler allows parties to execute Oblivious RAM programs directly, i.e., without first unrolling it into a circuit, which provided an alternative approach to secure RAM computation. The method was further improved by Gordon et al. [20] in order to perform sublinear amortized database search. Lu and Ostrovsky [25] considered two-server Oblivious RAM inside the Ostrovsky-Shoup compiler, which led to logarithmic overhead in both the computation and the communication complexity. Note that these three works allow secure RAM evaluation without having to unroll the program into a circuit and represent a different way to perform secure computation that reveals only the program running time. Among these, [25] is the best result for programs (instead of circuits) in terms of *computation complexity* and *communication complexity*. However, in terms of *round complexity*, these papers leave much to be desired: they all require at least logarithmic rounds for *each* CPU computation step. Even "non-interactive" ORAM [36] requires at least one round of interaction for each read and write. Thus, since the running time of CPU is at least t steps for programs that run in time t, this leads to $\Omega(t)$ round complexity using all previous methods. In contrast, in this paper we show how to retain poly-log overhead in communication and computation, and make the entire computation completely non-interactive in the OT-hybrid model, just like Yao.

1.1 The Blueprint for RAM Program Garbling

We describe our approach at a high level: we start with an ORAM compiler (with certain properties which we will describe later) that takes a program and converts it into an oblivious one. We call this new program the "ORAM CPU" because it can be thought of as a client running a CPU that performs a local computation followed by reading or writing something on the remote server. As a conceptual segue, consider the following change: instead of the ORAM CPU locally performing its computation, it creates a garbled circuit representing that computation, and also garbles all the inputs for that computation (the inputs are just the client state and the last fetched item, possibly with some randomness) and sends it to the server who then evaluates the circuit. The output of this computation is just the next state and the next read/write query, and the server preforms the read/write query locally, and sends back the result of the read/write query along with the state to the ORAM CPU. We emphasize that this is just a conceptual intermediate step, since this step does not actually give us any savings and possibly interferes with the security of the ORAM CPU by having its state revealed to the server.

Next, we change where the ORAM CPU state is stored: instead of letting the client hold it, it is stored on the server in garbled format. That is to say, the garbled circuit that the client sends to the server now outputs a garbled state instead of a regular state, which can then be used as input for the next ORAM CPU step. As long as the garbled circuit for the next CPU step uses the same input encoding as the one generated by our current CPU step, then the client does not need to interact with the server. However, the garbled CPU also performs read/write operations into ORAM memory that need to be carefully interleaved with our computations. We need to describe how this is done next.

Let us suppose that the ORAM compiler had the property that the ORAM CPU knows exactly when the contents of a memory location that it wants to read next was last written to (which is the case for many ORAM schemes). We attempt to perform the same strategy as we did with garbling the state: whenever the ORAM CPU wants to write something to memory. We store memory bits as Yao's garbled keys, based on the actual location, and the time last written. Thus, the bit stored in some particular location has one of the two garbled keys. However, this does not immediately work, because if each memory location uses a different encoding, the CPU circuit does not know which encoding to use when reading at some future time.

In order to resolve this, we construct a circuit that assists with this transition: the circuit takes as input a time step and memory location computes (in a garbled form) two possible encodings for 0/1 encoded in this location and outputs a garbled circuit encoded for that time step to "translate" keys stored in memory to keys needed by the CPU. Since this circuit does not require the knowledge of the memory location ahead of time, the client can generate as many of these as needed at the *start* of the computation. Indeed, if the ORAM program runs in t steps, the client can generate t of these circuits, garble them, and send them all to the server, non-interactively.

Note that we need Oblivious RAM with poly-log overhead where the client *size* is at most some fixed polynomial in the security parameter times some poly-log factor in n. This is because for every ORAM fetch operation, we also need to emulate the client's internal computation of the Oblivious RAM using garbled circuit, which incurs a multiplicative overhead in the size and the running time of the client. Thus, the smaller the client of Oblivious RAM, the more efficient our solution is: in order to achieve poly-log overhead, all Oblivious RAM schemes where client memory is larger than poly-logarithmic (e.g. [13, 7]) is not useful for our purposes. We expand on the intuition in Section 3.1. In Section 3 we give the main construction for garbled RAM programs. When combined with oblivious transfer, this gives a one-round secure two-party RAM program computation in the semi-honest model (which can be extended to multi-party using the Beaver-Micali-Rogaway paradigm[3]), which we discuss in Section 4. In Appendix B, we give a direct construction for a single-round ORAM. This follows from our general result, but in case where only single read/write is needed this gives a more direct construction which may be useful in some of the applications.

1.2 Related Work on Oblivious RAMs and Secure RAM Computation.

Oblivious RAM was introduced in the context of software protection by Goldreich and Ostrovsky [15]. In the original work by Goldreich [13], a solution was given with $O(\sqrt{n})$ and communication overhead where lookups could be done in a single round and $O(2^{\sqrt{\log n \log\log n}})$ communication overhead for a recursive solution. Subsequently, Ostrovsky [28, 29] gave a solution with only poly-log overhead and constant client memory (the so-called "hierarchical solution"). This solution, and most other subsequent solutions (except for two exceptions described below) require a logarithmic number of rounds per query. Let us consider a single round solutions only:

- The original single-round solution of Goldreich [13] requires $\sqrt{n}$ client's memory.
- Recently, Boneh et al. [7] extended Goldreich solution to work on larger blocks, but also requiring $\sqrt{n}$ client's memory.
- Williams [37] and Williams and Sion [36], presented a single-round Oblivious RAM that has logarithmic overhead, logarithmic client storage and uses bloom filters.
- Even more recently, Gentry et al. [12] proposed an extension of Sion-Williams scheme [36] that is also one-round.

In contrast to all of the above works, we propose the first solution for single-round ORAM solution that makes use of Yao's Garbled Circuits, is based on any one-way function (or any block cipher), using constant client memory in the security parameter and poly-logarithmic overhead.

Subsequent to Goldreich and Ostrovsky [28, 29, 13, 15], works on Oblivious RAM (e.g. [38, 36, 39, 31, 17, 18, 33, 19, 22, 34]) looked at improving the concrete and asymptotic parameters of Oblivious RAM. The works of Ajtai [1] and Damgård et al. [10] show how to construct oblivious RAM with information-theoretic security with poly-logarithmic overhead in the restricted model where the adversary cannot read memory contents. That is, these results work in a model where an adversary only sees the sequence of accesses and not the data. The notion of *Private Information Storage* introduced by Ostrovsky and Shoup [30] allows for private storage and retrieval of data, and was primarily concentrated in the information theoretic setting. This model differs from Oblivious RAM in the sense that, while the communication complexity of the scheme is sub-linear, the server performs a *linear* amount of work on the database. The work of Ostrovsky and Shoup [30] gives a multi-server solution to this problem in both the computational and the information-theoretic setting and introduces the Ostrovsky-Shoup compiler of transforming Oblivious RAM into secure RAM computation. The notion of single-server "PIR Writing" was subsequently formalized in Boneh, Kushilevitz, Ostrovsky and Skeith [6] where they provide a single-server solution. The case of amortized "PIR Writing" of multiple reads and writes was considered in [8].

With regard to secure computation for RAM programs, the implications of the Ostrovsky-Shoup compiler was explored in the work of Naor and Nissim [27] which shows how to convert RAM programs into so-called circuits with "lookup tables" (LUT). The Ostrovsky-Shoup compiler was further explored in the work of Gordon et al. [20] in the case of amortized programs. Namely, consider a client that holds a small input x, and a server that holds a large database D, and the client wishes to repeatedly perform private queries $f(x, D)$. In this model, an expensive initialization (depending only on D) is first performed. Afterwards, if f can be computed in time T with space S with a RAM machine, then there is a secure two-party protocol computing f in time $O(T) \cdot \mathsf{polylog}(S)$ with the client using $O(\log S)$ space and the server using $O(S \cdot \mathsf{polylog}(S))$ space. The secure RAM computation solution of Lu and Ostrovsky [25] can be viewed as a generalization of the [30] model where servers must also perform sublinear work.

1.3 Our Results

In this paper, we show how to garble any Random Access Machine (RAM) Program π_t that runs in time upper bounded by t while keeping all the non-interactive advantages of the Yao's Garbled Circuit approach. More specifically, we present a program garbling method which consists of a triple of polynomial-time algorithms (G, GI, GE). G takes as input any RAM program π_i that includes an upper bound t on its running time and a pseudorandom function (PRF) family F and a seed s for PRF of size k (a security parameter) and outputs a garbled program $\Pi_t = G(\pi_t, t, F, s)$, where all inputs are polynomial in the security parameter. Just like gabled circuits, we provide a way to garble any input x for π_t into Garbled Input $X = GI(x, s)$, and an algorithm to evaluate a garbled program on garbled inputs $GE(\Pi_t, t, X)$. The correctness requirement is that for any x, π_t, F, s it holds that $\pi_t(x) = GE(G(\pi_t, t, F, s), GI(x, s))$ with the security guarantee that nothing about x is revealed except its running time t, expressed in terms of computational indistinguishability ($\approx$) between the simulator Sim and garbled outputs. So far, the above description matches Yao's garbled circuit description. The difference is both in the running time and the size of garbled program for our new garbling method.

Main Theorem *Assume one-way functions exist, and let the security parameter be k and let F be a PRF family based on the one-way function. Then, there exists a Program Garbling triple of poly-time algorithms G, GI, GE such that for any t any π_t and any input x of length n we have the following.*

Correctness: $\forall x, \pi_t, F, s$: $\pi_t(x) = GE\left[G(\pi_t, t, F, s), GI(x, s)\right]$.

Security: $\exists$ *poly-time simulator Sim, such that $\forall \pi, t, x, s$, where $|s| = k$,*

$$[G(\pi_t, t, F, s), GI(x, s)] \approx Sim\left[1^k, t, |x|, \pi_t(x)\right].$$

Garbled Program Size: *The size of the garbled program*

$$|G(\pi_t, t, F, s)| = O\left((|\pi| + t) \cdot\ k^{O(1)} \cdot \textit{polylog}(n)\right).$$

Garbled Input Size: *Let $|x| = n$ and $|s| = k$. $\forall x, s$ the garbled input size*

$$|GI(x, s)| = O\left(n \cdot k^{O(1)} \cdot \textit{polylog}(n)\right).$$

Our main construction is a garbled program based on any one-way function (or a block-cipher), and is time-compact in the sense that if the original program runs in t time and has size n, our garbled RAM runs in $O(t \cdot poly(k, \log n))$.

1.4 Remarks

- **Making programs and outputs private.** We note that similar to Yao, we can make π_t to be a time-bounded **universal program** u_t, (i.e., an interpreter) and $x = (\pi'_t, y)$ include both time-bounded program π'_t and input y, so that $u_t(x) = \pi'_t(y)$. Part of the specification of π'_t may also include masking its output – i.e. to have output blinded (XORed) with a random string. That allows, just like Yao, to keep both the program and the output hidden from a machine that evaluates the garbled program. Such a modification has been utilized in the literature (see, e.g. [2]).

- **Reactive functionalities.** Our result shows that we can first garble a large input x, $|x| = n$ with garbled input size equal to $O(|x| \cdot k^{O(1)} \cdot \mathsf{polylog}(n))$ so that later, given private programs $\pi^1_{t_1}, \ldots, \pi^j_{t_j}, \ldots$ for polynomially many programs where program π^j runs in time t_j and potentially modifies x, (e.g., database updates) we can garble and execute all of these programs just revealing running times t_i, and nothing else. The size of each garbled program remains $O\left((|\pi^i| + t_i) \cdot k^{O(1)} \cdot \ \mathsf{polylog}(n)\right)$. It is also easy to handle the case where the length of x changes, provided that an upper bound by how much each program changes the length of x is known prior to garbling of next program.

- **Cloud computing.** As an example of the power of our result we outline secure cloud computation/delegation. In this simple application one party has an input and wants to store it remotely and then repeatedly run different private programs on this data. Reactive functionalities allow us to do this with one important restriction: we do not give the server a choice in adaptively selecting the inputs: but this is not an issue as the server itself has no inputs to the program. The other possible problem is if the programs themselves are contrived and circularly reference the code for the garbling algorithm. Such programs would be highly unnatural to run on data and so we disallow them in our setting.

- **Two-party computation.** Note that just like in Yao's garbled circuits, in order to transmit the garbled inputs corresponding to input bits held by a different party for the sake of secure two-party computation, one relies on Oblivious Transfer (OT) that can be done non-interactively in the OT-hybrid model. Here, we insist that the OT-selected inputs to our garbled program are committed to prior to receiving the RAM garbled program, i.e. non-adaptively [4].

- **Optimizations.** We remark that step two of our blueprint is applicable to almost all ORAM schemes with small CPU as follows: instead of collapsing in the hierarchical Oblivious RAMs multiple rounds of a single read/write to a single round, we can implement our step 2 directly for each round of each read/write (e.g. even inside a single read/write simulation of Oblivious RAM that requires multiple rounds) of the underlying Oblivious RAM: by implementing an oracle call for each Oblivious RAM CPU read/write using our method of compiling memory fetch "on the fly" into garbled circuits. Any Oblivious RAM where the CPU can tell precisely when any memory location was overwritten last can be complied using our approach. (We call such Oblivious RAMs "predictive memory" RAMs and explore this further in the full version.) For example, this property holds for [22] ORAM. It also allows a generic method to "collapse" all multi-round predictive memory Oblivious RAM with small CPU into a single round. Observe that the overall complexity for garbling programs depends both on the CPU complexity and the ORAM read/write complexity.

- **Tighter Input Compactness.** Using an ORAM scheme that has small input encoding and small size CPU (such as [22]) we can also make Input Compactness in our main theorem tighter: for all programs we can make garbled inputs to be $O(nk)$, where recall that n is the input size and k is the security parameter. We remark that if we wish to garble only "large" programs that run time at least $\Omega(n \cdot \log n \cdot k^{O(1)})$, we can make Input Compactness even better under the assumption that one can encode inputs to garbled circuits to be of size $O(n + k)$ and have the garbled program "unpack" the inputs to the full $O(nk)$ size. Such packing techniques for have been recently developed for garbling the inputs of garbled circuits by Ishai and Kushilevitz [21].

- **Stronger Adversarial models.** As already mentioned we describe the scheme in the honest-but-curious model based on honest-but-curious Yao, and only in the non-adaptively secure setting (see [4] for further discussion of adaptivity.) There is a plethora of works that convert Yao's garbled circuits from honest-but-curious to malicious setting, as well strengthening its security in various settings. Since our machinery is build on top of Yao's garbled circuits (and Obvious RAMs that work in the fully adaptive setting), many of these techniques for stronger guarantees for Yao's garbled circuit apply in a straightforward manner to our setting as well. We postpone description of malicious models to the full version.

2 Preliminaries

2.1 Oblivious RAM

We work in the RAM model with stored programs, where there is a CPU that can run a program that performs a sequence of reads or writes to locations stored on a large memory. This machine, which we will refer to as the CPU or the client, can be viewed as a stateful[1] processor with only a few special data registers that store program counters, query counters, and cryptographic keys (primarily a seed for a PRF) and that CPU can run small programs which model a single CPU step. Given the CPU state Σ and the most recently read element x, $CPU(\Sigma, x)$ does simple operations such as addition, multiplication, updating program counter, or executing PRF followed by producing the next read/write command as well as updating to the next state Σ'.

Because we wish to hide the type of access performed by the client, we unify both types of accesses into a operation known as a *query*. A sequence of n queries can be viewed as a list of (memory location, data) pairs $(v_1, x_1), \ldots, (v_n, x_n)$, along with a sequence of operations $op_1, \ldots, op_n$, where op_i is a READ or WRITE operation. In the case of READ operations, the corresponding x value is ignored. The sequence of queries, including both the memory location and the data, performed by a client is known as the *access pattern*.

In our model, we wish to obliviously simulate the RAM machine with a client, which can be viewed as having limited storage, that has access to a server. However, the server is untrusted and assumed to malicious. An oblivious RAM is *secure* if the view of a any malicious server can be simulated in poly-time in a way that is indistinguishable from the view of the server during a real execution.

Concretely, we focus on the hierarchical Oblivious RAM scheme of Ostrovsky [28, 29]. There is a data structure that consists of a sequence of buffers $B_k, B_{k+1}, \ldots, B_L$ of geometrically increasing sizes, e.g. B_i is of size 2^i. Typically $k = O(1)$ (the first buffer is of constant size) and $L = \log n$ (the last buffer may

[1] We can consider a *stateless* version where all registers are stored in memory. For ease of exposition, we let the client hold local state.

contain all n elements), where n is the total number of memory locations. For ease of exposition, we set $k = 1$ in the sequel. These buffers are standard bucketed hash tables, where each B_i consists of, say 2^i, buckets, each of size b. To read or write to a memory location v from the hierarchical data-structure, we wish to hide the identity of the buffer from which the element was found. Specifically, we start by reading the top (smallest) buffer B_1 in its entirety; then, for each $1 \leq i \leq L$, we compute $j = h_i(v)$ (where $h_i(\cdot)$ is a hash function implemented as a PRF with appropriate domain and range for each level) and read the entire j-th bucket (b elements) of that buffer. This alone is not sufficient, as if we make identical queries, the same locations will be scanned. Thus, once element v is found at some level, we search upon random dummy locations from subsequent (bigger) level buffers. In addition, at the end of this process, we re-insert element r (overwriting it in case of a write) into the top buffer of the data-structure. This, together with the re-shuffling procedure described below, guarantees that when executing future operations with the same v, independent locations will be read from each buffer. Finally, we remark that even if element v was not in any buffer before the operation, it will be inserted into the top buffer.

After every 2^i insertions, buffer B_i is considered "full" and its contents are moved into the next buffer B_{i+1}. More precisely, we do the following: after $m = 2^i \cdot \ell$ reads or writes, ℓ odd, where m is divisible by 2^i but not by 2^{i+1}, we move all the elements from buffers $B_1, \ldots, B_i$ into buffer B_{i+1} (at such time step, B_{i+1} itself is empty). For this, we pick a fresh pseudo-random hash function for B_{i+1} (which can be modeled using a PRF). Finally, there is a process called oblivious hashing which we will use in detail later that removes any correlation between the new locations of the elements in B_{i+1} and their old locations.

2.2 Yao's Garbled Circuits

Garbled circuits were introduced by Yao [40]. A series of works looked at proving the security and formalizing the notions of garbled circuits, including Lindell and Pinkas [23], and recently, the work of Bellare et al. [5]. We refer the reader to the latter work for more details, and we briefly summarize the key properties.

A circuit garbling scheme we view as a triple of algorithms (G, GI, GE) where $G(1^k, C)$ takes as input a security parameter k and circuit C and outputs some garbled circuit Γ and garbling key gsk. $GI(x, gsk)$ converts an input x and a gsk into a garbled input X, and $GE(\Gamma, X)$ evaluates a garbled circuit on an garbled input.

We also paraphrase a summary of garbled circuits given in Choi et al. [9] which uses a point-and-permute variant due to Malkhi et al. [26]. Consider a circuit C and a CPA-secure symmetric encryption scheme $(KeyGen, Enc, Dec)$. For each wire $i \in C$, we generate two random keys w_i^0, w_i^1 and a random bit "flip" indicator π_i. We associate w_i^b with $\lambda_i^b = b \oplus \pi_i$ and call the pair $(w_i^0||\lambda_i^0, w_i^1||\lambda_i^1)$ the 0 and 1 labels for wire i respectively (the λ will be henceforth omitted in following sections). For each fan-in 2 gate g with input wires i, j and output wire k, we associate a garbled table to the gate consisting of the following four ciphertexts:

$$
\begin{aligned}
&EnC_{w_i^{\pi_i}}\left(EnC_{w_j^{\pi_j}}\left(w_k^{g(\pi_i,\pi_j)}||\pi_k \oplus g(\pi_i,\pi_j)\right)\right)\\
&EnC_{w_i^{\pi_i}}\left(EnC_{w_j^{\neg\pi_j}}\left(w_k^{g(\pi_i,\neg\pi_j)}||\pi_k \oplus g(\pi_i,\neg\pi_j)\right)\right)\\
&EnC_{w_i^{\neg\pi_i}}\left(EnC_{w_j^{\pi_j}}\left(w_k^{g(\neg\pi_i,\pi_j)}||\pi_k \oplus g(\neg\pi_i,\pi_j)\right)\right)\\
&EnC_{w_i^{\neg\pi_i}}\left(EnC_{w_j^{\neg\pi_j}}\left(w_k^{g(\neg\pi_i,\neg\pi_j)}||\pi_k \oplus g(\neg\pi_i,\neg\pi_j)\right)\right).
\end{aligned}
$$

Given this garbled table and labels for the wires i and j ($w_i^{b_i}||\lambda_i^{b_i}$ and $w_j^{b_j}||\lambda_j^{b_j}$ respectively), a party can decrypt the row corresponding to $\lambda_i^{b_i}, \lambda_j^{b_j}$ to obtain the proper label for the output wire: $w_k^{g(b_i,b_j)}||\lambda_k^{b_k}$. If the labels to the input wires are given, then one can recursively evaluate all gates of the circuit. Suppose now the two parties wish to securely evaluate C on input (x, y) where the circuit generator holds x and the circuit evaluator holds y. The circuit generator sends the labels for the wires corresponding to the proper bits of x, and the labels for the input wires corresponding to the proper bits of y can be sent using oblivious transfer. The circuit evaluator at the end will receive a bunch of labels containing the λ_o for all the output wires o.

We use a variant of Yao's garbled circuits in which some of the output wires are revealed to the circuit evaluator immediately in the clear and some are not revealed. Revealing output wires in the clear is the standard way of viewing garbled circuits. For output wires that are not revealed, they are either represented as internal garbled keys (that can be used as inputs for other circuits) or XORed with pseudo-random pads that can later be revealed. It will be clear from the context which representation of various outputs we use.

We first make an observation that the labels (keys) on a given wire used in a garbled circuit can be re-used in additional newly generated gates, as long as the value does not change between the uses and it is not revealed whether this label represents 0 or 1. (For example, assume that garbled circuit evaluator is given a label on some input wire, which is a key representing a 0 or a 1. We claim that the same key can be used as input key for other garbled circuits that are generated later.) This observation allows us to execute garbled circuits in "parallel" or "sequentially" where some labels are re-used. Indeed, this observation is implicitly used in classic garbled circuits in gates where the fan-out is greater than 1: all outgoing wires share the same labels (see e.g. Footnote 8 in Lindell-Pinkas [23]).

Lemma 1. *Suppose $\mathcal{C}$ and $\mathcal{C}'$ are two circuits and suppose there is some input x for which we want to compute $\mathcal{C}(x)$ and $\mathcal{C}'(x)$ (resp. $\mathcal{C}(\mathcal{C}'(x))$). Suppose the wires $w_0, \dots, w_n$ in $\mathcal{C}$ represent the input wires for x and similarly define $w'_0, \dots, w'_n$ represent the input wires of x in $\mathcal{C}'$ (resp. $v'_0, \dots, v'_n$ be the output wires of $\mathcal{C}'$). Let $k^b_{w_i}$ represent the label indicating wire $w_i = b$, and let C and C' be randomly garbled into $GC(\mathcal{C})$ and $GC(\mathcal{C}')$ under the restriction that $k^b_{w_i} = k^b_{w'_i}$ (resp. $k^b_{w_i} = k^b_{v'_i}$). Then the tuple $(GC(\mathcal{C}), GC(\mathcal{C}'), \{k^{x_i}_{w_i}\}_{i=0}^n)$ can be computationally simulated.*

Proof. Consider the composite circuit $D = \mathcal{C}||\mathcal{C}'$ (resp. $E = \mathcal{C} \circ \mathcal{C}'$) which is just a copy of $\mathcal{C}$ and a copy of $\mathcal{C}'$ in parallel (resp. sequence). Then every garbling of D induces a garbling of $\mathcal{C}$ and $\mathcal{C}'$ with the restriction exactly as above. By the security of garbled circuits, there exists a simulator that can simulate $(GC(D), \{k^{x_i}_{w_i}\}_{i=0}^n)$. We can construct a simulator for our lemma by simply taking this simulator and taking the output and separate out $GC(\mathcal{C})$ and $GC(\mathcal{C}')$, as the lemma requires. $\square$

Remark: If the data is encrypted bit by bit using Yao's keys, Lemma 1 allows us to run arbitrary garbled circuits on this data, akin to general purpose "function evaluation" on encrypted data. This observation itself has a number of applications, we describe these in the full version of the paper.

3 Non-interactive Garbled RAM Programs

3.1 Informal description of main ideas

We consider the RAM model of computation as in the works of [15, 28, 29] where a RAM program along with data is stored in memory, and a small, stateful CPU with a $O(1)$ instruction set that can store $O(1)$ words that can be of size $\mathsf{polylog}(n) = poly(k)$ where k is the security parameter. Our starting point is a ORAM model that can tolerate fully malicious *tampering* adversary (see [29, 15]). Each step of the

CPU is simply a read/write call to main memory followed by executing its next CPU instruction. We now summarize our ideas for building Garbled RAM programs from an Oblivious RAM program.

In order to garble a RAM program π_t, we consider the two fundamental operations separately and show how to mesh them together:

1. Read/Write (v, x) from/to memory.
2. Execute an instruction step to update state and produce next read/write query: $\Sigma', \text{READ/WRITE}(v', x') \leftarrow CPU(\Sigma, x)$. Updating the state can include updating local registers, incrementing program counters and query counters, and updating cryptographic keys.

Our goal is to transform this into a *non-interactive* process by letting the client send the server enough garbled information to evaluate the program up to t steps, where t upper bounds the RAM program running time. We give some intuition as to how to construct a circuit for each step, and then how to garble them. The first part will be modeled as the circuit $\mathcal{C}_{ORAM}$, and the second part will be modeled as the circuit $\mathcal{C}_{CPU}$. The circuits satisfy a novel property: the *plain circuit* $\mathcal{C}_{ORAM}$ emulates a query for the ORAM client and outputs a bit representation of a garbled circuit GC_{ORAM}. This GC_{ORAM} has output encodings that will be compatible with the *garbled circuit* $GC(\mathcal{C}_{CPU})$ to evaluate a garbled the CPU's next step. We remark that GC_{ORAM} actually contains several sub-circuits, but is written as a single object for ease of exposition. If we generate t of these garbled circuits, then a party can evaluate a t-time garbled RAM program by consuming one garbled $\mathcal{C}_{ORAM}$ and one garbled $\mathcal{C}_{CPU}$ per time step.

We first consider the circuit $\mathcal{C}_{CPU}$, which is straightforward to describe. This circuit takes as input Σ representing the internal state of the CPU, and x the last memory contents read. Recall that the CPU performs a step $CPU(\Sigma, x)$ and updates the state to Σ' and gives the next read/write query to memory location v' and contents x'. In order to turn this into a circuit, we can sacrifice some efficiency and have a "universal" instruction in which we run *every* atomic instruction (from its constant sized instruction set) and simply multiplex the actual results using the instruction opcode. This universal instruction is modeled as a circuit which is of size $k^{O(1)}$. We remark that although this circuit is simple, the complexity arises from when we want to garble this circuit: the garbling must be done in a way so that the garbled inputs and outputs are compatible with GC_{ORAM}.

The circuit $\mathcal{C}_{ORAM}$ must emulate the client in Oblivious RAM (we can think of it as being a non-interactive client either by breaking out each individual step as a separate circuit, or using a non-interactive ORAM). The input of the circuit is just an ORAM read/write query[2], and the output of the circuit is **a bit representation that describes a set of garbled circuits, equivalent to what would have been produced via the ORAM client** which we call GC_{ORAM}.[3] We give full details on the construction in Section 3.2. It is important that we argue that the result of this fetch can be combined with the evaluation of the CPU step. Observe that since the labels in our single-round ORAM are generated as pseudo-random time-labeled encodings, so we know ahead of time only the encoding of the output (but know neither the input nor output) of the i-th invocation of the single-round ORAM. Thus when garbling $\mathcal{C}_{CPU}$, the input encodings use exactly the output encodings from the respective outputs of the ORAM. Recall in our single-round ORAM protocol the server sends back the encoded output to the client; here, we *do not* send it back, and instead keep the result and use it as input in the next CPU step (which is secure and correct via Lemma 1).

[2]Since the ORAM client uses randomness as well as time-labeled encodings (which are outputs of the PRF), we will allow these to be *inputs* to $\mathcal{C}_{ORAM}$, so that they may be pre-computed "for free" rather than computed via the circuit. The circuit consumes these inputs in order to generate the output garbled circuit *without* having to evaluate these itself. The only thing the circuit does not have ahead of time is the hash of the location of the query at each level, so our circuit $\mathcal{C}_{ORAM}$ must use PRFs to compute them.

[3]GC_{ORAM} consists of a set of $|B_1| + 2L - 2$ garbled $GC(\mathcal{C}_{match})$, corresponding garbled $GC(\mathcal{C}_{next})$, a garbled $GC(\mathcal{C}_{write})$, and all necessary time-dependent updates $GC(\mathcal{C}_{update})$ as in Theorem 2.

Then, putting it all together, to garble a RAM program π_t that runs in time t, the program garbling algorithm G generates t garbled $\mathcal{C}_{ORAM}$ and $\mathcal{C}_{CPU}$ circuits, and also encodes the initial state Σ_0 of the CPU with the program initialized, counters set to zero, and with fresh cryptographic keys. The full construction of G is given the next section, Section 3.2.

Looking ahead, in the context of secure two-party computation, this garbled program can be sent to the server in a single round, whereupon the server can evaluate the program by itself. The result is sent back to the client, and since the labels were all generated pseudo-randomly, the client can determine whether the output bits are zero or one. In the case where the server also has inputs, the client can generate the pseudo-random labels and then the server uses Oblivious Transfer to select the ones corresponding to its input. We mention that in the OT-hybrid model, this is a non-interactive protocol, we can avoid adaptivity issues by requiring the server to provide its inputs upfront at the same time the client sends its garbled program, i.e. this can be viewed as just a one-step process where the garbled program is sent "along" with the garbled inputs via the OT functionality.

3.2 Main Construction of Garbled Programs

We first describe how to construct the algorithms G, GI, GE. Given a program π_t running in time t, we describe the algorithm G that converts it into a garbled program Π_t. In order to do so, we follow the two steps outlined above and we consider the construction of a circuit that performs an ORAM query $\mathcal{C}_{ORAM}$ and a circuit that runs one CPU step $\mathcal{C}_{CPU}$.

Our garbling algorithm G will provide enough garbled circuits to execute t steps of a program π_t. Each step is a garbled RAM query (done obliviously via our single-round ORAM) followed by a garbled CPU computation. It starts with a garbled encoding of the initial state Σ_0 of the CPU with the program π_t initialized, counters set to zero, and with fresh cryptographic keys. For each of the t time steps, it creates a garbled $GC(\mathcal{C}_{ORAM})$ for a read/write of that time step, then a garbled $GC(\mathcal{C}_{CPU})$ to perform a CPU step. We show how to construct $\mathcal{C}_{ORAM}$ and $\mathcal{C}_{CPU}$ such that they can be garbled and interleaved. We will show that this garbling is independent of the actual program path, regardless of what memory locations have been fetched, and is correct and secure.

First, we describe $\mathcal{C}_{ORAM}$ to mimic an oblivious read/write access to main memory. For this, it can just perform the steps in our single-round Oblivious RAM, with one difference: G does not know ahead of time which memory location will be used. Hence, in order to overcome this, the circuit $\mathcal{C}_{ORAM}$ must take a memory location *as input* and internally formulate what the ORAM client computes. $\mathcal{C}_{ORAM}$ outputs what the "virtual" ORAM client would have sent to the server: a garbled circuit GC_{ORAM} representing a read/write query. The novelty in this construction is that when we feed a memory location v into $\mathcal{C}_{ORAM}$, the output precisely is a garbled ORAM read/write query relative to that memory location. In order to hide v, both $\mathcal{C}_{ORAM}$ and v are garbled into $GC(\mathcal{C}_{ORAM})$ and V respectively, and by the correctness of garbled evaluation, the output is still GC_{ORAM}. By the security of the underlying ORAM, this output GC_{ORAM} can actually be simulated.

Although it is a circuit that outputs another circuit, there is no circularity in this construction: given a query location and some fixed randomness, the behavior of the ORAM client is completely deterministic, straight-line, and takes $k^{O(1)} \cdot \mathsf{polylog}(n)$ steps, so the output can be represented by a circuit also of that size. This ORAM client is independent of the main program CPU which only uses ORAM as an "oracle". We emphasize this again, because G will most likely be ran by a client, G does not play the role of the ORAM client but rather *emulates* the ORAM client via $\mathcal{C}_{ORAM}$, so this is *not* a client attempting to capture its own logic in a circuit. We provide a pseudocode description of $\mathcal{C}_{ORAM}$ in Figure 1.

Looking ahead, G will garble this circuit and ensure that the output of an ORAM query has the same

Inputs: An ORAM query to read/write (v, x) and a query number ℓ. This circuit interprets the client performing the ℓ-th ORAM query, which uses randomness and time-labeled encodings based on ℓ. As such, this circuit also takes these randomness bits and pre-computed encodings as inputs.
Output: A garbled circuit GC_{ORAM} representing a read/write ORAM query.
Circuit Description: We describe the functionality of the circuit $\mathcal{C}_{ORAM}$. We recall our algorithm for a ORAM query. Using time-labeled encodings via PRFs, it generates a set of $|B_1| + 2L - 2$ garbled $GC(\mathcal{C}_{match})$ which has hard-coded location information built into it, with corresponding garbled $GC(\mathcal{C}_{next})$ circuits, and one final $GC(\mathcal{C}_{write})$ garbled circuit for writing the element back to the top level (and possibly an update circuit). Although the ORAM client evaluates these PRFs internally, we *do not* encode this as part of our circuit $\mathcal{C}_{ORAM}$, but rather we "consume" them as input. Similarly, the ORAM client must use randomness, which we also consume from the input of $\mathcal{C}_{ORAM}$. Since the circuit itself emulates the ORAM client during a query, it appears similar to the construction in Figure 5, but with the key difference that the encodings/PRFs are fed as inputs.

1. For the top level, B_1, for each bucket, $\mathcal{C}_{ORAM}$ creates a time-labeled garbled circuit $GC(\mathcal{C}_{match})$ consuming the input encodings to be used as garbled labels.

2. For subsequent levels $i = 2 \ldots L$:

   (a) The circuit $\mathcal{C}_{ORAM}$ computes $q_i^0 = h_i(v)$ and consumes q_i^1 from the input (the input itself is uniformly random)

   (b) Consume two secret keys for encryption sk_i^0 and sk_i^1 from the input and create a garbled circuit $GC(\mathcal{C}_{next})$

   (c) Create two time-labeled garbled circuits $GC(\mathcal{C}_{match})$, one that searches for w in bucket q_i^0 encrypted under sk_i^0, and one that searches for w in bucket q_i^1 encrypted under sk_i^1, again consuming the encoding from the input to $\mathcal{C}_{ORAM}$.

3. $\mathcal{C}_{ORAM}$ also creates a garbled $GC(\mathcal{C}_{write})$ that writes the result back to the first empty position the top level buffer B_k.

4. If ℓ is a multiple of $|B_1|$, then a reshuffle step is performed using the time-labeled garbled update circuit $GC(\mathcal{C}_{update})$.

5. The combined set of garbled circuits is referred to as GC_{ORAM}.

We point out that throughout this entire process, every time a query circuit is created, G increments ℓ in order to keep track of the time-labeled encodings required by the $\mathcal{C}_{ORAM}$ circuits.

Figure 1: The ORAM Client Circuit $\mathcal{C}_{ORAM}$

encoding as that used to garble $\mathcal{C}_{CPU}$. The algorithm G can then garble both $\mathcal{C}_{CPU}$ and $\mathcal{C}_{ORAM}$ ahead of time, without having to know the memory location.

Next, we consider building the circuit which performs a single CPU step in the RAM program, $\mathcal{C}_{CPU}$ that is supposed to perform $\Sigma', \textsc{Read}/\textsc{Write}(v', x') \leftarrow CPU(\Sigma, x)$. In order to hide which instruction is being executed, we build the circuit to take an instruction opcode and we run every single-step instruction *from its constant sized instruction set (not all possible program paths)* of the CPU. The circuit multiplexes the actual results using the instruction opcode. This universal instruction is modeled as a circuit which is of size $k^{O(1)}$ and is independent of the ORAM circuit, independent of the queried locations, and independent of the current running time.

One may ask the question: How can this circuit be interleaved with the $\mathcal{C}_{ORAM}$ circuit if it is independent of it?

The answer is that when G garbles $\mathcal{C}_{CPU}$, the encoding will depend on the output of $\mathcal{C}_{ORAM}$ in the previous time-step. Note that this construction is not circular as each garbling only depends on the previous one, leading up to a total of t time steps. This can be done because G knows the encoding of the *output encoding* (but not the output) of the Oblivious RAM query, which *does not depend* on the location queried. This output encoding is then used for the input parameter encoding for $GC(\mathcal{C}_{CPU})$. We provide a pseudocode description of G in Figure 2.

Inputs: A program π_t with an upper bound on running time t, and a pseudo-random function family F along with a key s.
Algorithm Description: The algorithm G is performed as follows. It creates an encoding of the initial state of the CPU, Σ_0 with the program π_t initialized. It also encodes an initial program counter and cryptographic keys. We show how to construct $\mathcal{C}_{ORAM}$ and $\mathcal{C}_{CPU}$ such that they can be garbled and interleaved across t time steps. We must argue that this garbling is independent of the actual program path, regardless of what memory locations have been fetched, and is correct and secure.
For each time step $i = 1 \ldots t$, G creates:

1. A garbled read/write query circuit $GC(\mathcal{C}_{ORAM})$ for performing query number i on some (unknown variable) garbled location V_i (and X_i in the case of a write). G pre-computes randomness and PRF evaluations and hardwires them. Although G does not know the eventual output, it knows the *encoding* of it, which is *independent of the queried location*. It uses this encoding for the following:

2. A garbled instruction circuit $GC(\mathcal{C}_{CPU})$ with input wires of X_i using the encoding from above, and the input wires of Σ_i using the output encoding from the previous CPU step. The output is a garbled location V_{i+1} (and X_{i+1} in the case of a write) to be used in the next read/write query and an garbled updated state Σ_{i+1}.

Figure 2: Program Garbling Algorithm G

The algorithm GI for garbling an input of size n is just the time-labeled encodings starting from wherever the RAM program expects the inputs to be located.

The algorithm GE used to evaluate a garbled program Π_t on garbled inputs evaluates the garbled circuit $GC(\mathcal{C}_{ORAM})$, then executing the garbled instruction $GC(\mathcal{C}_{CPU})$ one at a time, up to t times. The process is precisely performing the same steps as G except evaluating garbled circuits instead of generating them. In addition, once it gets the garbled ORAM query, it must also execute it as well. We provide a pseudocode description of G in Figure 3.

3.3 Main Result

We now state our main result:

Theorem 1. *Assume one-way functions exist, and let the security parameter be k and let F be a PRF family based on the one-way function. Then, there exists an efficient Program Garbling triple of algorithms G, GI, GE such that for any π_t any t and any input x of length n, we have the following.*

Inputs: A garbled program Π_t with garbled input X.
Algorithm Description: The algorithm GE is performed as follows. It first stores the initial encoded program state and inputs into memory. Then, for each time step $i = 1 \dots t$, GE performs:

1. Evaluate the garbled query circuit $GC(\mathcal{C}_{ORAM})$ on a garbled memory location V_i. The output is GC_{ORAM} which itself is a garbled circuit that represents a read/write query in our single-round ORAM protocol. Execute the query playing the role of the server to obtain some garbled output X_i which is kept locally instead of sent to the client.

2. Evaluate the garbled instruction circuit $GC(\mathcal{C}_{CPU})$ on garbled inputs X_i and Σ_i. Obtain a new read/write query V_{i+1}.

After t steps, output the final value X_{t+1}.

Figure 3: Garbled Program Evaluation Algorithm GE

Correctness: $\forall x, \pi_t, F, s$:
$\pi_t(x) = GE\left[G(\pi_t, t, F, s), GI(x, s)\right]$.
Security: $\exists$ *poly-time simulator* Sim*, such that* $\forall \pi, t, x, s$*, where*
$|s| = k \left[G(\pi_t, t, F, s), GI(x, s)\right] \approx Sim\left[1^k, t, |x|, \pi_t(x)\right]$.
Program Size: *The size of the garbled program*
$|G(\pi_t, t, F, s)| = O\left((|\pi| + t) \cdot k^{O(1)} \cdot \textsf{polylog}(n)\right)$.
Input Size: *Let* $|x| = n$ *and* $|s| = k$*.* $\forall x, s$ *the garbled input size*
$|GI(x, s)| = O\left(n \cdot k^{O(1)} \cdot \textsf{polylog}(n)\right)$.

Proof.

Correctness. This construction is correct due to the correctness of the underlying single-round Oblivious RAM scheme in Theorem 2 and the correctness of garbled circuits. In addition, we need to verify that when *interleaving* the garbled instruction execution along with the ORAM fetch queries, the ability to properly decrypt and evaluate the garbled circuits is maintained. Because G generates a garbled circuit $GC(\mathcal{C}_{ORAM})$ to simulate the fetching client inside the ORAM, the output encoding is chosen so that it matches the input encoding of $GC(\mathcal{C}_{CPU})$. Thus, since G generates sufficiently many circuits for ORAM fetches corresponding to the i-th instruction executed (with respect to time, regardless of the ordering of the actual instructions in π), the GE algorithm evaluating the garbled circuits can properly evaluate the instruction and throw away any unused fetches corresponding to the i-th step.

Security. In order to show security, we must show that there exists a simulator Sim that can simulate the garbled execution given only the running time and program output. In order to do so, we consider what a server running the algorithm GE does during the execution of the garbled program.

It first stores the initial encoded program state and inputs into memory. Then, for each time step $i = 1 \dots t$, GE performs: In each CPU step of the garbled program execution, the server performs the following:

1. Evaluate the garbled query circuit $GC(\mathcal{C}_{ORAM})$ on a garbled memory location V_i. The output is GC_{ORAM} which itself is a garbled circuit that represents a read/write query in our single-round ORAM protocol.

2. Execute the garbled ORAM query GC_{ORAM} playing the role of the server to obtain some garbled output X_i which is kept locally instead of sent to the client.

3. Evaluate the garbled instruction circuit $GC(\mathcal{C}_{CPU})$ on garbled inputs X_i and Σ_i. Obtain a new read/write query V_{i+1}.

By Theorem 2, the underlying single-round Oblivious RAM is secure and uses time-labeled garbled circuits and encodings and can be simulated by Sim_{ORAM}. Furthermore, the underlying Yao's garbled circuits are secure, and can be simulated by Sim_{Yao}. Thus, the access pattern of the ORAM can be simulated even for tampering adversary, and we need only show that the garbled circuit emulating the ORAM client $GC(\mathcal{C}_{ORAM})$ and garbled instructions $GC(\mathcal{C}_{CPU})$ can also be simulated. The garbled circuits can be interleaved securely due to Lemma 1, and the time-labeled encodings themselves are just outputs of a PRF. By the security of Yao's garbled circuits and the underlying PRF, these can be simulated securely.

Program Size. We analyze the cost of garbling a program. First, to garble all the instructions of the program, we incur a cost of $O(|\pi| \cdot k^{O(1)} \cdot \mathsf{polylog}(n))$. Furthermore, because the overhead of our underlying ORAM is $k^{O(1)} \cdot \mathsf{polylog}(n))$ and since at each time step, the client must prepare "CPU instruction" circuits which include some constant number of ORAM queries, we incur another $O(t \cdot k^{O(1)} \cdot \mathsf{polylog}(n))$. Overall this leads to the garbled program being of size $O((|\pi| + t) \cdot k^{O(1)} \cdot \mathsf{polylog}(n))$.

Input Size. We analyze the cost of garbling an input of size n. Each bit of the input is encoded and stored in the ORAM hierarchy which incurs a $O(k^{O(1)} \cdot \mathsf{polylog}(n))$ multiplicative overhead, the total size of the garbled input is therefore $O(n \cdot k^{O(1)} \cdot \mathsf{polylog}(n))$.

□

4 Application to Secure RAM Computation

We give an example application in which only one party has input and wants to repeatedly run programs on this data. Such is the case of secure cloud computing, where someone stores data in the cloud and then later runs computations against that data. We emphasize that in this setting, there is no issue of adaptivity because the server has no inputs. In the typical setting of two-party secure computation, we deal with this by making the server first perform OTs to retrieve its inputs *before* the client sends the garbled program. In the multi-party setting, the technique can be utilized in the Beaver-Micali-Rogaway paradigm [3] to achieve constant-round MPC with the same approach as in [3] but with garbled RAM programs.

That is to say, in this application, a client wishes to store some data x on a remote server and then run various RAM programs on x without the server learning the results of the programs or x itself. Of course, the client could always ignore the server altogether and run all the programs on x locally, so we are envisioning a scenario in which the client does not want to carry around all of its data locally and wants to only store a few cryptographic keys or counters. To apply Garbled RAM programs to this application, the client first garbles the input x to get $X = GI(x)$ and sends it to the server. Then for each program the client wants to run, it recalls the encoding of the previous output and creates a garbled program using the labels of the previous output as inputs for the current program.

5 Conclusions and Open Problems

Recently, Goldwasser at. al. [16] have shown how to construct a *reusable* Garbled Yao. It is tempting to plug it into our construction to achieve reusable GRAM with compactness proportional to program size and independent of its running time. The idea is to compute poly-many iterations of the CPU computation using reusable Yao (instead of sending fresh garbled circuit for each CPU step) where CPU computes its own garbled keys for each step. This is possible only if there exists poly-time reusable circular-secure Garbled Yao with input encoding of size independent of the circuit size. Constructing such a gadget is an interesting open problem even under non-standard assumptions.

6 Acknowledgements

We thank Oded Goldreich and Daniel Wichs for very helpful discussions and the anonymous reviewers for their comments.

References

[1] Miklós Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In *STOC*, pages 181–190, 2010.

[2] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *ICALP (1)*, pages 152–163, 2010.

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

[4] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT*, pages 134–153, 2012.

[5] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *ACM Conference on Computer and Communications Security*, pages 784–796, 2012.

[6] Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E. Skeith III. Public key encryption that allows PIR queries. In *CRYPTO*, pages 50–67, 2007.

[7] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. CSAIL Technical Report, MIT-CSAIL-TR-2011-018, 2011.

[8] Nishanth Chandran, Rafail Ostrovsky, and William E. Skeith III. Public-key encryption with efficient amortized updates. In *SCN*, pages 17–35, 2010.

[9] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the ”free-xor” technique. In *TCC*, pages 39–53, 2012.

[10] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.

[11] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[12] Craig Gentry. Personal communication, 2012.

[13] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987.

[14] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

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

[16] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Succinct functional encryption and applications: Reusable garbled circuits and beyond. Cryptology ePrint Archive, Report 2012/733, 2012.

[17] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, pages 576–587, 2011.

[18] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *CCSW*, pages 95–100, 2011.

[19] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *SODA*, pages 157–167, 2012.

[20] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security*, pages 513–524, 2012.

[21] Yuval Ishai and Eyal Kushilevitz. Personal communication, 2012.

[22] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156, 2012.

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

[24] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. Cryptology ePrint Archive, Report 2012/601, 2012.

[25] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *TCC*, pages 377–396, 2013.

[26] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302, 2004.

[27] Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, pages 590–599, 2001.

[28] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523, 1990.

[29] Rafail Ostrovsky. *Software Protection and Simulation On Oblivious RAMs.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, June 1992.

[30] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[31] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519, 2010.

[32] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[33] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[34] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[35] Daniel Wichs. *Personal Communication*. March 2013.

[36] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *ACM CCS*, pages 293–304, 2012.

[37] Peter Williams. *Oblivious Remote Data Access Made Practical*. PhD thesis, SUNY Stony Brook, Dept. of Computer Science, 2012.

[38] Peter Williams and Radu Sion. Usable PIR. In *NDSS*, 2008.

[39] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*, pages 139–148, 2008.

[40] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.

A Glossary of Circuits

Circuit	Description
$\mathcal{C}_{CPU}$	Circuit for evaluating a CPU step.
$\mathcal{C}_{ORAM}$	Mimics an ORAM client query, outputting garbled versions of the above circuits.
GC_{ORAM}	Garbled circuit that is the output of $\mathcal{C}_{ORAM}$. Consists of garbled circuits used in single-round ORAM:
$\mathcal{C}_{match}$	Matches a memory location in a bucket.
$\mathcal{C}_{next}$	Outputs next bucket to probe depending on found/not found.
$\mathcal{C}_{update}$	Performs oblivious hashing for ORAM update.
$\mathcal{C}_{write}$	Writes output to top level buffer.

Figure 4: Glossary of Circuits

B Single-Round Oblivious RAM From Any One-way Function

B.1 Informal Description of Main Ideas

As a starting point, we consider the hierarchical ORAM of Ostrovsky [28, 29] and use the same terminology as in Ostrovsky's Ph.D. thesis [29]. In this scheme, the data is encrypted (under semantically secure private-key encryption) and stored in hierarchical levels that reshuffle and move into larger levels as they fill up. To keep track of the movement, each level is temporally divided into different time periods called *epochs*, based on how many queries the client has already performed. The client only needs to keep track of the keys corresponding to the latest epoch for each level, which in turn only depends on the total number of queries that so far have been performed.

In our new solution, we maintain the same hierarchical levels, but encrypt all bits within the level differently. To explain our encryption method, we first generalize Pseudo-Random Functions (PRF) into a multi-argument PRF $F_s(x_1, x_2, ..., x_{k-1}, x_k)$ which is computationally indistinguishable from a truly random multi-argument function. Our multi-argument PRF, instead of outputting a single bit, outputs a pseudo-random key of length proportional to the security parameter, i.e. a sufficiently long key for a private-key encryption scheme. Such a multi-argument PRF can be trivially constructed from any standard PRF.

We now describe the encoding of each bit in the hierarchical solution of Oblivious RAM that we use. For each bit in each buffer of some level we can uniquely define its location by epoch number, level number, bucket number within the level and address within the bucket. Let us call these specifications $x_1, \ldots, x_{k-1}$ (where the details of the encoding will be specified later). Now we define two keys for each such bit: $F_s(x_1, \ldots, x_{k-1}, 0)$ and $F_s(x_1, \ldots, x_{k-1}, 1)$. One key corresponds to "encoding of zero" and the other corresponds to "encoding of one". Jumping ahead, we will use these encodings inside multiple Yao's garbled circuits repeatedly using Lemma 1. More specifically, for every buffer bit b we encode it as $F_s(x_1, \ldots, x_{k-1}, b)$ and write this key into the buffer as the encryption of this bit. We remark that since keys are generated pseudo-randomly, the client will not need to remember anything except the PRF keys used to generate the labels and the current epoch. We describe at a high level how an ORAM read/write is performed in a single round, and how a re-shuffle is performed.

Recall that in the hierarchical ORAM scheme, to fetch a (virtual) memory location v, the client first scans the entire top buffer B_1 in its entirety, then, until v is found in some bucket, computes the hash $j = h_i(v)$ and looks up bucket j for each subsequent level B_i for $i \leq L$. Once v is found, the client retrieves a random bucket j in subsequent levels. If we let q_i denote the bucket that is fetched at level i for $k + 1 \leq i \leq L$, then the important observation in [37, 36] is that there are only two "choices" for q_i: $h_i(v)$ or random. Thus, even though the choice of which q_i to use is done interactively, the client can pre-compute a list of $2L - 2$ buffer addresses (two for each level, except the smallest first level which is accessed in its entirety) of the form $((h_2(v), r_2), \ldots, (h_L(v), r_L))$.

The way we encode all values within each buffer allow us to prepare $2L - 2$ garbled circuits that operate as follows. We prepare a circuit that reads the smallest level, since the inputs are keys to the garbled circuit. The circuit checks if the value is there or not, and depending if it is found or not "decrypts" one of the two circuits for the next level, which also indicates which buffer in the next level the circuit is prepared for. That is, for each level we prepare two garbled circuits, one to access random buffer (if the value already found) and one to access the location where value could be located. We encrypt both circuits using private key encryption. Each circuit outputs a decryption key for the next circuit, as well as the buffer number that the just-decrypted circuit is prepared for. The circuit that reads the smallest level is given un-encrypted.

Additional book-keeping is done to pass information between different circuits and to execute the last circuit that re-writes the top-level buffer. We stress that the labels in those garbled circuits are generated pseudo-randomly and depend only on the epoch as well as former inputs. When evaluating the ORAM query, the server evaluates a garbled circuit for each level in the hierarchy in turn, which allows him to decrypt the next circuit and tells it the location to apply in the next level buffer the just decrypted circuit.

Finally, we observe that oblivious updates/re-shuffling can be done through garbled circuits implementing sorting networks, where this can be done through several invocations of the sorting network. We now proceed to give a more detailed description.

Toward this end, we give two definitions which will help in our construction. We recall the notion of a time-labeled RAM simulation due to Ostrovsky [28, 29]: after any number of queries-so-far, there exists a way for the client to efficiently compute the number of times it has previously accessed a particular memory location. As briefly explained above, we define the notion of a so-called *time-labeled encoding* (via Yao garbling) so that whatever is stored in memory, the client can efficiently compute the encoding of it.

Definition 1. *Suppose some element* (v, x) *was stored in level* B_i *in bucket* j *in some position* ℓ *inside the bucket during epoch* e*. We define the* time-labeled encoding *of* (v, x) *to be a bit-by-bit encoding where the b-th bit of* (v, x) *is encoded as* $F_s(i, j, \ell, e, b)$ *where* F *is a multi-argument PRF with output being a random element in the keyspace of our symmetric-key encryption scheme.*[4]

Next, consider how the client will build a garbled circuit whose input is in some specific buffer in the memory hierarchy. If the stored information is a time-labeled encoding, the client can compute two pseudorandom keys for each bit stored in the buffer, where one of the keys is a "zero" key and another key is a "one" key. The client knows that one of the two keys is stored in the buffer representing either the encoding of zero bit or one bit in that location. Hence, one can construct a garbled circuit operating on the buffer using Lemma 1. The client prepares two garbled circuits for each level as described before. It also encrypts both circuits with a private-key encryption where exactly one key will be revealed depending on whether or not v has been found already or not. We keep track if the item has been found or not and depending on this variable, we release to the server a decryption key for one of the two circuits for the larger level, together with is buffer address. Finally, we need to write the found element back to the top level buffer, and possibly perform a hierarchy update.

Definition 2. *Suppose some element* (v, x) *was stored in level* B_i *in bucket* j *in some position* ℓ *inside the bucket during epoch* e*. Let* C *be a circuit operating on this bucket. Let* sk *be a secret key to some encryption scheme. We define the* time-labeled garbling of C *to be a garbled circuit* $GC(C)$ *with* special encodings of the wires *in which the labels corresponding to the wires of the hash buckets are precisely the time-labeled encoding of elements in the buckets. The entire circuit is then encrypted under* sk*.*

B.2 Single-Round Oblivious RAM Construction

As building blocks for our construction, we give details on the circuits which we described above. The circuits are building blocks to perform the following procedures: We have one circuit searches for a memory location in a bucket and returns the memory contents and keeps track of whether or not it is found. We have one circuit that tells the server where to look in the next level: it returns either the real hash location or a random location (and the corresponding decryption key) depending on the found variable. We have one circuit that writes the final answer to the top level buffer. We have one circuit to update the ORAM hierarchy.

Here in the details, we highlight a key difference between the utility of $\mathcal{C}_{match}$ and $\mathcal{C}_{next}$: the output of the first circuit, when garbled, will be encoded and obscured from the server, but the output of the second circuit,when garbled, will be released to the server in the clear since it must fetch and decrypt. It will be convenient to refer to them by name, so we describe them in more detail. More formally, the logic of the circuits are given as:

1. $\mathcal{C}_{match}$: It takes as input an some hash location j in level B_i, a (virtual) memory location w that the client is searching for, a storage variable y and an indicator bit $found$. The logic is that if w has already been found, do nothing, otherwise attempt to find (v, y) where $w = v$ and store it into y. It outputs y' and $found'$ with the following behavior:

[4]Specifically, from any one-way function, one can build PRFs and CPA-secure symmetric-key encryption scheme that satisfy this requirement.

$$\begin{cases} y' = y, found' = 1 & \text{if } found = 1; \\ y' = x, found' = 1 & \text{if } found = 0 \text{ and there is some } (v, x) \text{ in the bucket such that } v = w; \\ y' = y, found' = 0 & \text{otherwise.} \end{cases}$$

2. $\mathcal{C}_{next}$:It takes as input an indicator bit $found$, some level i and two bucket locations q^0 and q^1 on level B_i, and two secret keys for encryption sk_0 and sk_1. It outputs q^{found} and sk_{found}.[5]

3. $\mathcal{C}_{write}$: Given a memory location w and memory contents y, it encodes (w, y) relative to the first empty slot in the top level buffer.

4. $\mathcal{C}_{update}$: Although oblivious hashing is described as an interactive process in most ORAM schemes, observe that in many cases (again, e.g. [15]) it ultimately amounts to performing many steps of a large parallel sorting and re-labeling algorithm wherein the elements need to be retrieved, decrypted, and re-encrypted. Instead, we can represent this as a large parallel circuit, using sorting networks to perform the sorting, and using time-labeled encodings instead of encryption. (More generally, the updates have fixed memory access that can be revealed to the circuit evaluator and are executed in strait-line. Any such program can be converted to garble circuit directly.)

We describe our construction relative to any hierarchical ORAM scheme that uses hash tables, though concretely one can think of the Goldreich-Ostrovsky [15] scheme. For a client to read/write to a memory location w, the client computes garbled buffer search circuits (i.e. $GC(\mathcal{C}_{match})$) for the top level B_1. Then the client pre-computes the hash of w for each of the levels $B_2, \ldots, B_L$, i.e. it sets $q_i^0 = h_i(w)$. It also generates $L-1$ random locations $q_i^1 = r_i$ for each level. This gives the client a list of $L-1$ pairs of bucket locations, 2 for each level (one real, one random): $((q_2^0, q_2^1), \ldots, (q_L^0, q_L^1)) = ((h_2(w), r_2), \ldots, (h_L(w), r_L))$.

For each of these $2L-2$ locations, the client makes a time-labeled garbled circuit that searches for w (i.e. it creates $GC(\mathcal{C}_{match})$ for those locations) and encrypts them under brand new (pseudo-randomly generated) encryption key sk_i^j. It is the case that only sk_i^j decrypts the garbled circuit for location q_i^j. In order to ensure that the server only gets the correct location and key to go from a level to the next (depending on found/not found), we must rely on the circuit $\mathcal{C}_{next}$ that produces exactly one out of the two location/key pairs for each level. In order to do so, the client hardwires locations q_i^0, q_i^1 and keys sk_i^0, sk_i^1 into generates a garbled $GC(\mathcal{C}_{next})$ that outputs where to go depending on found/not found and only the correct sk_i^j. Finally, the client creates a time-labeled garbled circuit $GC(\mathcal{C}_{write})$ that "writes" a time-labeled encoding of (w, y) back to the top level buffer B_1 (i.e. it re-writes the entire top level). To perform hierarchy updates, it uses a garbled $GC(\mathcal{C}_{update})$. The full client and server construction is given in Figure 5.

B.3 Analysis

The goal of this section is to show the following theorem:

Theorem 2. *Assume one-way functions exist. Then the construction of given in Figure 5 is a secure single-round Oblivious RAM with* $\mathsf{polylog}(n) \cdot k^{O(1)}$ *overhead with client only needing* $k^{O(1)}$ *memory to store the cryptographic keys.*

[5]This can be thought of as a circuit for obliviously transferring one-out-of-two of the locations and keys.

Client performing a read/write to memory location w with the read/written value being y:

1. For each bucket in the top level, B_1, the client creates a time-labeled garbled circuit $GC(\mathcal{C}_{match})$ that searches for w. The circuits are constructed so that the output encodings matches the input encodings in the subsequent circuit (i.e. circuit chaining as in Lemma 1).
2. Pre-compute all hash locations $q_i^0 = h_i(w)$ and and random locations $q_i^1 \leftarrow 0 \dots |B_i|$ for levels $i = 2 \dots L$. Pseudo-randomly generate secret keys for encryption sk_i^0 and sk_i^1.
3. For subsequent levels $i = 2 \dots L$:
   (a) Create a time-labeled garbled $GC(\mathcal{C}_{next})$ by hardwiring q_i^0, q_i^1, sk_i^0, and sk_i^1 as inputs, the only free variable being the $found$ flag. The outputs are *unencoded*. The labels for $found$ should match the $found$ output from the previous level $i - 1$.
   (b) Create two time-labeled garbled $GC(\mathcal{C}_{match})$ circuits, one that searches for w in bucket q_i^0 and one that searches for w in bucket q_i^1. Encrypt the first under under sk_i^0, and encrypt the second under sk_i^1.
4. Create a time-labeled garbled $GC(\mathcal{C}_{write})$ that takes the final output y (or y from the write query) and writes it back to the first empty position the top level buffer B_1.
5. The client in one round sends all these circuits to the server, then the client receives the final output y and decodes it.
6. The client increments the local query counter t. If t is a multiple of $|B_1|$, then a reshuffle step is performed using the time-labeled garbled update circuit $GC(\mathcal{C}_{update})$.

As for the server, it performs the following steps:

1. Receive all the garbled circuits from the client.
2. It evaluates $GC(\mathcal{C}_{match})$ for every bucket in the top level B_1.
3. For subsequent levels $i = 2 \dots L$:
   (a) Evaluate $GC(\mathcal{C}_{next})$ with the garbled found/not found flag from the previous level $i - 1$ and obtain in the clear a location q_i and a key sk_i.
   (b) On bucket q_i, decrypt the $GC(\mathcal{C}_{match})$ using sk_i and evaluate it, keeping track of the garbled $found'$ flag and garbled memory contents y'.
4. Evaluate the last garbled circuit $GC(\mathcal{C}_{write})$ which outputs some time-labeled encoding of (w, y) and store it in the first empty position in B_k and send the encoded output y to the client.
5. In case of an update, evaluate $GC(\mathcal{C}_{update})$ and rewrite the relevant levels of the hierarchy with the corresponding time-labeled output.

Figure 5: Single-Round ORAM

Proof.

Correctness. This construction is correct due to the correctness of the underlying ORAM scheme and the correctness of garbled circuits. The only additional step we need to check is that the output of one circuit correctly feeds into the input of the other. Because the labels for the relevant wires are actually time-labeled encodings, they are correct by the way they are constructed due to Lemma 1.

Cost Analysis. We analyze the cost in terms of communication, computation, and rounds for both the client and the server. The round complexity is clearly 1. The client must create $|B_1| + 2(L-1) = O(\log n)$ garbled circuits for $\mathcal{C}_{match}$ and $\mathcal{C}_{next}$ and one final garbled circuit for writing the element back to the top level. The sizes of $\mathcal{C}_{match}$ and $\mathcal{C}_{next}$ are both $O(\mathsf{polylog}(n))$. It must also garble $\mathcal{C}_{update}$ which is of size $O(|B_i| \cdot \mathsf{polylog}(n))$ every $|B_i|$ steps. Each element of the underlying ORAM scheme is now encoded bit-by-bit where each bit now turns into $k^{O(1)}$ bits which is size of the output of our multi-argument PRF. The PRF is evaluated at most twice per wire of each garbled circuit, and the underlying encryption scheme is evaluated at most eight times per gate of each circuit, each of these invocations run in $k^{O(1)}$. Since the underlying ORAM scheme only has poly-log overhead, the overall computation and communication for the client amounts to $\mathsf{polylog}(n) \cdot k^{O(1)}$ per query. The server has the same communication complexity, and the computation is just the evaluation of the garbled circuits, which amounts to at most four decryptions per garbled gate, thus also resulting in $\mathsf{polylog}(n) \cdot k^{O(1)}$.

Security. In order to show security, we must show that there exists a simulator Sim that generates the view of the server for a sequence of polynomially sized t queries. First, we generate a simulated garbled circuit for the top level. By the security of garbled circuits, there exists some simulator Sim_{Yao} that simulates these garbled circuits (except we use the time-labeled encodings of the inputs and outputs in the simulation, which can further be simulated by true randomness due to the security of our PRF).

Next, we describe how to create a good simulation of the subsequent levels. Sim has to simulate $2L-2$ (encrypted) $GC(\mathcal{C}_{match})$ circuits which produce garbled outputs, and more problematically, Sim must simulate $GC(\mathcal{C}_{next})$ circuits which produces outputs *in the clear*. In order to do so, we rely on the fact that the locations given in the clear can be simulated in turn. Indeed, by the security of our underlying ORAM, there exists a simulator Sim_{ORAM} that generates the access pattern of the ORAM across all t queries. This access pattern gives us a list of locations in the intermediate buffers $\ell_2^i \in B_2, \ldots, \ell_L^i \in B_L$ for each query $i = 1 \ldots t$. Our simulator Sim will use these locations as the simulated outputs of the garbled $GC(\mathcal{C}_{next})$ circuits. To simulate the view of the server seeing the output of $GC(\mathcal{C}_{next})$ (on the j-th level in the i-th query), we set it to be the simulated location ℓ_j^i and a randomly chosen secret key sk_i which can only decrypt the proper circuit in the next level. These we simulate via Sim_{Yao} given only the output ℓ_j^i, sk_i. Also, because the encrypted garbled $GC(\mathcal{C}_{match})$ circuits for these locations will be decrypted, we can also simulate them via Sim_{Yao}. For the remaining locations that won't be decrypted, our simulator Sim encrypts the "all-zeroes" string, which is computationally indistinguishable from a good encryption.

Finally, by Lemma 1, we can reuse the encodings as inputs between different invocations while still being able to simulate.

$\square$