How to Garble RAM Programs[3]

Steve Lu[1] and Rafail Ostrovsky[2]

[1] Stealth Software Technologies, Inc. steve@stealthsoftwareinc.com
[2] Department of Computer Science and Department of Mathematics, UCLA. Work done while consulting for Stealth Software Technologies, Inc. rafail@cs.ucla.edu
[3] Patent Pending

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.

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 [33] or the Goldreich-Micali-Wigderson [10] 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 [8,27]: Given a T-time RAM program, [8] shows how one can transform it into a $O(T^3)$-time TM, and [27] 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 [10] 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 [25] which used Oblivious RAM [11] 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. [16] in order to perform sublinear amortized database search. Lu and Ostrovsky [21] 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, [21] 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 one round for *each* CPU computation step, even using the so-called non-interactive RAM solution of [31], which reduces each read/write to one round between the client and the server. 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. In contrast, in this paper, we show how to retain poly-log overhead in communication and computation, and make the entire computation 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. [9,6]) 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[2]), which we discuss in Section 4. In the full version [20], we also show how to construct a single-round ORAM.

1.2 Related Work on Secure RAM Computation.

Oblivious RAM was introduced in the context of software protection by Goldreich and Ostrovsky [11]. In the original work by Goldreich [9], 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 [23,24] gave a solution with only poly-log overhead and constant client memory (the so-called "hierarchical solution").

Subsequent to Goldreich and Ostrovsky [23,24,9,11], works on Oblivious RAM (e.g. [31,32,26,13,14,28,15,18,29]) looked at improving the concrete and asymptotic parameters of Oblivious RAM. The notion of *Private Information Storage* introduced by Ostrovsky and Shoup [25] 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 [25] 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 [5] where they provide a single-server solution. The case of amortized "PIR Writing" of multiple reads and writes was considered in [7].

With regard to secure computation for RAM programs, the implications of the Ostrovsky-Shoup compiler was explored in the work of Naor and Nissim [22] 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. [16] 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 [21] can be viewed as a generalization of the [25] 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. [1]).
- **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 [3].

- **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 [18] 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 [18]) 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 [17].
- **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 [3] 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[4] processor with only a few special data registers that store program counters,

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

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. For concreteness, we focus on sequence of buffers $B_k, B_{k+1}, \ldots, B_L$ of geometrically increasing sizes. Typically $k = O(1)$ (the first buffer is of constant size) and $L = \log n$ (the last buffer may contain all n elements), where n is the total number of memory locations. These buffers are standard bucketed hash tables, with buckets of size b. We refer the reader to [11] for more information.

2.2 Yao's Garbled Circuits

Garbled circuits were introduced by Yao [33]. A series of works looked at proving the security and formalizing the notions of garbled circuits, including Lindell and Pinkas [19], and recently, the work of Bellare et al. [4]. 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 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 [19]).

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 $w_0, \ldots, w_n$*

in $\mathcal{C}$ represent the input wires for x and similarly define $w'_0, \ldots, w'_n$ represent the input wires of x in $\mathcal{C}'$ (resp. $v'_0, \ldots, v'_n$ 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 $(GC(\mathcal{C}), GC(\mathcal{C}'), \{k^{x_i}_{w_i}\}^n_{i=0})$ 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 $(GC(D), \{k^{x_i}_{w_i}\}^n_{i=0})$. 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.

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 [11,23,24] 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 [24,11]). 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: $\Sigma', \textsc{Read/Write}(v', x') \leftarrow CPU(\Sigma, x)$. 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[5], 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}. 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 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 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 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).

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.

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 $\mathcal{C}_{CPU}$.

[5] 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.

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 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 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 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 $\Sigma', \textsc{Read/Write}(v', x') \leftarrow CPU(\Sigma, x)$. 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?

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 $\mathcal{C}_{ORAM}$.

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.

Fig. 1. The ORAM Client Circuit $\mathcal{C}_{ORAM}$

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.

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

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}.

Fig. 2. Program Garbling Algorithm 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.

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 $i = 1 \ldots t$, 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 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}.

Fig. 3. Garbled Program Evaluation Algorithm GE

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.*

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.
We give an outline of the proof of security, and refer the reader to the full version [20] for the full proofs.

Security. We design the simulator Sim as follows. We know that 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 the underlying ORAM.
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}.

The underlying Oblivious RAM is secure and uses time-labeled garbled circuits and encodings and can be simulated by some Sim_{ORAM}. Furthermore, the underlying Yao's garbled circuits are secure, and can be simulated by some 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.

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 [2] to achieve constant-round MPC with the same approach as in [2] 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. [12] 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 Acknolwedgements

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

References

1. Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *ICALP (1)*, pages 152–163, 2010.
2. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, pages 503–513, 1990.
3. 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.
4. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *ACM Conference on Computer and Communications Security*, pages 784–796, 2012.
5. Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E. Skeith III. Public key encryption that allows PIR queries. In *CRYPTO*, pages 50–67, 2007.
6. Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. CSAIL Technical Report, MIT-CSAIL-TR-2011-018, 2011.
7. Nishanth Chandran, Rafail Ostrovsky, and William E. Skeith III. Public-key encryption with efficient amortized updates. In *SCN*, pages 17–35, 2010.
8. Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.
9. Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987.
10. 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.
11. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.
12. 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.
13. Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, pages 576–587, 2011.
14. 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.
15. 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.
16. 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.
17. Yuval Ishai and Eyal Kushilevitz. Personal communication, 2012.
18. 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.
19. Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.
20. Steve Lu and Rafail Ostrovsky. How to garble RAM programs. Cryptology ePrint Archive, Report 2012/601, 2012.
21. Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *TCC*, pages 377–396, 2013.
22. Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, pages 590–599, 2001.

23. Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523, 1990.
24. Rafail Ostrovsky. *Software Protection and Simulation On Oblivious RAMs.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, June 1992.
25. Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.
26. Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519, 2010.
27. Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.
28. 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.
29. Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *NDSS*, 2012.
30. Daniel Wichs. *Personal Communication*. March 2013.
31. Peter Williams and Radu Sion. Single Round Access Privacy on Outsourced Storage. In *ACM CCS*, pages 293–304, 2012.
32. 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.
33. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.