Outsourcing Private RAM Computation

Craig Gentry* Shai Halevi † Mariana Raykova ‡ Daniel Wichs§

February 27, 2014

Abstract

We construct the first schemes that allow a client to privately outsource arbitrary program executions to a remote server while ensuring that: (I) the client's work is small and essentially independent of the complexity of the computation being outsourced, and (II) the server's work is only proportional to the run-time of the computation on a *random access machine (RAM)*, rather than its potentially much larger circuit size. Furthermore, our solutions are non-interactive and have the structure of *reusable garbled RAM programs*, addressing an open question of Lu and Ostrovsky (Eurocrypt 2013). We also construct schemes for an augmented variant of the above scenario, where the client can initially outsource a large private and persistent database to the server, and later outsource arbitrary program executions with read/write access to this database.

Our solutions are built from *non-reusable garbled RAM* in conjunction with new types of *reusable garbled circuits* that are more efficient than prior solutions but only satisfy weaker security. For the basic setting without a persistent database, we can instantiate the required reusable garbled circuits using *indistinguishability obfuscation*. For the more complex setting with a persistent database we need stronger notions of obfuscation. Our basic solution also requires the client to perform a one-time preprocessing step to garble a program at the cost of its RAM run-time, and we can avoid this cost using stronger notions of obfuscation. It remains an open problem to instantiate these new types of reusable garbled circuits under weaker assumptions, possibly avoiding obfuscation altogether.

1 Introduction

Outsourcing computation from a weak client to a more powerful server is quickly becoming the predominant mode of day-to-day computation, bringing with it new security challenges and flourishing research into methods for addressing them. In this work we consider the challenge of *private outsourcing*, where the client wants to execute a program on a remote server while hiding from it the raw data to be used in the computation. Moreover, we want to ensure that:

1. The client should perform significantly less work than executing the program, and
2. The server should not have to do much more work than executing the program.

One method of outsourcing computation relies on fully homomorphic encryption (FHE) [RAD78, Gen09], where the client simply encrypts her input and decrypts the output, and the server computes the program on encrypted data. Unfortunately, this solution requires the server to translate the program into a circuit and therefore work as hard as the *circuit size* of the computation, which in general, can be much larger than the work needed to execute the program on a random-access

*IBM Research, T.J. Watson. E-mail: cbgentry@us.ibm.com.

†IBM Research, T.J. Watson. E-mail: shaih@alum.mit.edu.

‡SRI. E-mail: mariana@cs.columbia.edu. Research conducted in part while at IBM Research.

§Northeastern University. E-mail: wichs@ccs.neu.edu. Research conducted in part while visiting IBM Research.
Supported in part by NSF grant 1347350.

machine (RAM). In particular, even if we reach the zenith of FHE efficiency, with no overhead per homomorphic addition/multiplication, simply converting the computation into a circuit may already be too inefficient.

In general, a RAM computation with run-time T can have Turing-Machine run-time and circuit size as high as $\widetilde{O}(T^2)$, which is already a considerably large overhead [CR73, PF79]. However, this distinction hardly tells the whole story, and the gap can be significantly larger in a setting involving program executions over a large persistent memory (e.g., a database). Consider for example the setting of private information retrieval (PIR) [CKGS98], where a server holds a large database of size N and a client wants to simply retrieve a single record from that database without the server learning the requested record. In this case, the RAM complexity of the retrieval query can be as low as $O(\log N)$, but the circuit complexity must be $\Omega(N)$ since the circuit must at least get the entire database as input, *a fully exponential gap*. The same gaps also already appear if we were to consider the Turing-Machine run-time of the computation instead of its circuit size.

We therefore would like to find an outsourcing protocol in which the server's work is only related to the RAM complexity of the program, while the client's work is essentially independent of the complexity of the program altogether. Furthermore, we would like to have such protocols in a setting where the client can initially outsource a large persistent memory (e.g., containing a database) and later outsource various RAM computations with read/write access to this memory.

Prior to this work, no such protocols were known. Although we do have private computation protocols over an outsourced memory based on *oblivious RAM* (ORAM) (e.g., [GO96, OS97, GKK$^+$12]) where the server's work is proportional only to the RAM complexity of the computation, in all of these protocols the client also works as hard as the server. In particular, these protocols allow the client to save on storage by outsourcing the data to a remote server, but they do not provide *any* savings in computation over executing the program locally on local data.

In this work we describe *reusable garbled RAM* schemes, which offer the first solution to private outsourcing of RAM computation, where the server's work is only proportional to the RAM run-time of the computation and the client's work is essentially independent of the complexity of the computation altogether. In addition, these protocols are *non-interactive*, i.e., they only use one-way communication if the server is to learn the output (or two message communication if the client is to learn the output), making them useful even beyond outsourcing in "send and forget" settings.

1.1 Garbled Circuits and Garbled RAM

(Reusable) Garbled Circuits. Garbled circuits, introduced in the seminal work of Yao [Yao82], allow a client to garble a circuit C and then an input x in such a way that a server can use these garbled values to compute $C(x)$ without learning anything more about x. Until recently all the schemes that we had became insecure if the server ever got to see more than one garbled input per garbled circuit. Last year Goldwasser et al. described the first *reusable* circuit-garbling scheme [GKP$^+$13b] where the client can garble a single circuit and then garble many inputs to that circuit without losing security. In this solution the client only needs to do a one-time pre-processing step to garble the circuit, at a cost proportional to the circuit size, but can then outsource many computations of this circuit on many different inputs by garbling them, with essentially no additional work per computation beyond what is needed to send the input. In [GKP$^+$13a], even the pre-processing step is removed for computations with a short Turing-Machine description. However, in both cases, the server still must work proportionally to the circuit size (or per-instance Turing Machine run-time) of the program for every evaluation, rather than the potentially much smaller RAM complexity.[1]

[1]The work of [GKP$^+$13a] also constructs attribute-based encryption for RAM programs, but this scheme does not hide the input over which the RAM computation is performed. It cannot be used in the context of outsourcing

Garbled RAM. Also recently, Lu and Ostrovsky introduced the notion of *garbled RAM* [LO13a]. Similar to garbled circuits, the client can garble a RAM program P, and later garble an input x in such a way that a server can use these garbled values to compute $P(x)$ without learning anything more about x. The complexity of garbling a RAM program (client complexity), the size of the garbled RAM, and the complexity of evaluating a garbled RAM (server complexity) are all proportional to the RAM run-time of the program rather than its circuit size.[2] The constructions of garbled RAM uses a clever combination of Yao garbled circuits and oblivious RAM (ORAM). Just like in Yao's circuits, the scheme is *not* reusable and becomes completely insecure if the server sees more than a single garbled input per garbled program. In other words, the client has to garble a fresh program for every computation, which requires as much work as doing the computation and therefore does not offer any savings in the context of outsourcing.

However, garbled RAM does offer an opportunity for amortization in the more complex setting involving multiple program executions over some persistent memory (e.g., a large outsourced database). The client can garble the initial memory contents once, and then can garble many different RAM programs and inputs (one input per program) that would be executed relative to the garbled memory, updating the memory with every execution. This property is called *persistent memory*, and allows the garbled memory to be *reused*. For example, the client can garble a large database of size N only once in time $O(N)$, and after that garble arbitrary queries to the database where the client work (time to garble a query) and the server work (time to evaluate a garbled query) are both proportional to the RAM run-time of the query. This provides a good solution for cases where the memory is large and the client wants to save on storage by outsourcing the contents, but the database queries are sufficiently simple that the client does not mind doing the work of the computation. It improves on simply using ORAM by making the program executions completely non-interactive. However, it still provides no savings in terms of client computation over having the client store the data and perform all computations locally.

Can Garbled RAM be Reusable? The above raises the natural question whether we we can obtain a *reusable garbled RAM*. In such scheme, the client can garble a program once as a potentially expensive pre-processing step, and later outsource many arbitrary computations of this program to a server by efficiently garbling fresh inputs. The server can evaluate the garbled program on a garbled input in time proportional to the RAM complexity of the program. Furthermore, we would also like to do this in a setting where the client initially garbles a large *persistent memory* (e.g., database) and the programs can read/write to this memory. Such reusable garbled RAM schemes give a particularly nice solution to the problem of *outsourcing private RAM computation* with no interaction in the case where the server is to learn the output, and one round of back-and-forth communication when the client is to learn the output. The output can be made private from the server by simply garbling an augmented program that returns the output encrypted under the client's key. Furthermore, it can be made *verifiable* so that the client can be sure that the received output is correct, by simply garbling an augmented program that returns the output along with a message-authentication-tag of the output, under a key provided with the input.[3]

private RAM computation.

[2] It was recently observed that the security proof for the scheme of [LO13a] has a subtle flaw, but the scheme can be fixed so as to get essentially the same properties as the original scheme [GHL$^+$14].

[3] We note that there are other approaches to verifiable RAM computation using SNARKs and proof-carrying data [Val08, BCCT13, BSCGT13, BSCG$^+$13, BFR$^+$13], but no other prior approaches that provide privacy. Therefore, we view the question of privacy as more pressing, but note that reusable garbled RAM gives us verifiability for free.

1.2 Our Solutions

In this work we describe the first solutions to the above problem of *reusable garbled-RAM*. We give three solutions with various tradeoffs between features/efficiency and the security assumptions needed to instantiate them.

As our **"basic"** solution, we describe a protocol that works in the setting *without* persistent memory, and requires the client to perform an expensive one-time pre-processing step to garble the program. In particular, the client can take a RAM program P with a run-time bound T and create a garbled version by working in time $\widetilde{O}(T)$. It can then very efficiently garble arbitrarily many inputs x_j to that program in time only proportional to the input (and output) size of the program, but independent of its complexity T. The server can evaluate the garbled program on each garbled input in time $\widetilde{O}(T)$. Furthermore, garbling new inputs only requires a *public key*, so anybody can outsource computations by creating garbled inputs to the garbled program.

As our **"best-case"** solution, we describe a protocol that also works in the more complex setting involving persistent memory (e.g., database) and does not require any expensive pre-processing. Specifically, in this solution the client has the option to garble some persistent memory of size N in time $\widetilde{O}(N)$. It can then garble a RAM program P in time proportional to its description length $O(|P|)$ but *independent of its running time*. Finally it can garble many "short inputs" x_i to the program P in time proportional to the input (and output) size of the program. The server can evaluate the garbled program with the garbled input over the garbled memory in time proportional to the program's RAM run-time $\widetilde{O}(T)$. For example, the programs P could be a SQL database implementation and the inputs x_i could specify various complex database queries. We stress that the program executions can both read and write to memory, and that the changes to memory made by one program execution persist for the next program execution and cannot be "rolled back" by a malicious server.

We also describe a **"middle"** solution that works in the setting of persistent memory just like the "best-case", but still requires expensive pre-processing to garble the program. In particular, garbling a RAM program P with RAM run-time T takes time $\widetilde{O}(T)$ time.

Assumptions. Our main contribution is to reduce the complex problems of reusable garbled *RAM* to seemingly simpler problems dealing with reusable garbled *circuits*, and avoiding the complexity of RAM altogether. Each of the above three constructions corresponds to a new notion of security/efficiency for reusable garbled circuits, which may be of independent interest (see below). Ultimately, we can instantiate these new notions of reusable garbled circuits using various obfuscation-based assumptions. The garbled circuits that are used in our "basic" solution can be based on indistinguishability obfuscation, while the ones needed for our "middle" and "best-case" solutions can be based on stronger variants of obfuscation, related to differing-inputs obfuscation. We stress that the use of obfuscation does not seem inherent, and there is hope that these new notions of reusable garbled circuits could be instantiated under simpler assumptions that avoid obfuscation altogether.[4]

1.3 Our Techniques

We obtain reusable garbled RAM from a combination of non-reusable garbled RAM, and a new form of reusable garbled circuits whose properties we discuss shortly.

[4]This is perhaps analogous to the question of indistinguishability-based security for functional encryption with unbounded collusion, where currently the only solutions we have are based on obfuscation [GGH+13], but it does not seem that obfuscation is inherent to the problem.

Our solutions are based on a very simple intuitive idea: given a RAM program P, consider the circuit $C[P]$ which has P hard-coded in its description, gets as input (r, x), and uses r as randomness to created a one-time garbled program $\tilde{P}_{one}$ (garbling P) and a garbled input $\tilde{x}_{one}$ (garbling x). Garbled RAM ensures that the circuit-size of $C[P]$ is only dependent on the RAM run-time T of the program P rather than its potentially much larger circuit size. Our main idea is to create a reusable garbled circuit $\tilde{C}_{reuse}$ of the circuit $C[P]$, which the client gives to the server. Each time the client wants to run a new program execution with input x_i, she chooses some fresh randomness r_i, and garbles (r_i, x_i) under the reusable circuit garbling scheme. The server runs the reusable garbled circuit $\tilde{C}_{reuse}$ on the garbled input from the client to *create* a one-time garbled RAM program $\tilde{P}_{one}$ and garbled input $\tilde{x}_{one}$, and then *evaluates* $\tilde{P}_{one}$ on $\tilde{x}_{one}$.

The above idea is not entirely new, but it turns out that it cannot quite work right out of the box.[5] Notice that the circuit $C[P]$ above has a huge output of size $\widetilde{O}(T)$ even though its input is small. Unfortunately, the reusable garbled circuit construction of Goldwasser et al. [GKP$^+$13b], requires that the size of the *garbled input* to the circuit always exceeds the size of the circuit's *output*, even if the size of the *actual input* of the circuit is small.[6] We call this property *output-size dependence*. In particular, to securely garble a short input (r_i, x_i), the client would have to create a huge garbled input of size $\widetilde{O}(T)$, which would require that the client works at least as hard as evaluating the program, and completely obliterate the efficiency benefits of outsourcing. Unfortunately, this is also not an accidental property of the construction of [GKP$^+$13b] and we show that *any* reusable circuit garbling scheme with simulation-based security must have output-size dependence (see Appendix C).

Our main observation is that we do not necessarily require full simulation-based security from the reusable garbled circuit component, even though we insist on achieving full simulation-based security for the final reusable garbled RAM construction. We come up with new notions of security for reusable garbled circuits that we call "distributional indistinguishability" (with several flavors), which turn out to suffice in our constructions and may be of independent interest elsewhere. Intuitively, these notions say that one cannot distinguish garbled inputs from two distributions that produce indistinguishable outputs. Moreover, these weaker security notions seem to plausibly allow for more efficient constructions that avoid "output-size dependence". Indeed, we propose new candidate constructions of such reusable garbled circuits based on obfuscation. Our three constructions of reusable garbled RAM translate to progressively more demanding flavors of security/efficiency for reusable garbled circuits with "distributional indistinguishability", which in turn translate to stronger notions of obfuscation needed to instantiate them. It remains as an open problem to achieve these notions from other assumptions, ideally avoiding obfuscation altogether.

1.4 Extensions

In Section 6 we explore several extensions and applications of our main results. We discuss how to generically augment reusable garbled RAM to get *output privacy* (server does not learn the output of the computation) and *verifiability* (client can be certain that the received output is correct). We also discuss how to get *program privacy* where the server does not learn the code of the program. Furthermore, we discuss how to leverage our solutions to get *input-specific run-time* where the server's work is only proportional to the RAM run-time of $P(x)$ on the desired input x rather than the worst-case run-time of P on inputs of size n. Lastly, we discuss applications to MPC where only one party needs to work as hard as the program's RAM run-time.

[5] A variant of an idea along these lines appeared in an early version of [LO13a] and was outlined in a rump-session talk [LO13b] but was retracted for exactly the reasons we describe here.

[6] The scheme and parameters of [GKP$^+$13b] are described for circuits with 1-bit output, but can easily be extended to the setting of multi-bit output at the above cost of having the size of the garbled-input grow with the output size.

2 Preliminaries

The two models of computation that we deal with in this work are circuits and RAM programs. Intuitively, a RAM program has access to some memory of size N and each step of the program can read/write to an arbitrary location of memory. We usually assume that the memory starts out empty. However, when we consider program executions over a persistent memory/database, it is useful to consider the case where the memory initially contains some data D. We use the notation $P^D(x)$ to denote the execution of a program P with random-access memory containing D and a short input x. For the RAM programs we consider in this work, we assume that we have an absolute bound on their worst-case running time, input/output length, and memory usage. A somewhat more detailed specification of the RAM model is found in Appendix B.1.

We use $C[\mathsf{prm}]$ or $P[\mathsf{prm}]$ to denote a circuit/program that depends on a parameter prm. The parameter can be an arbitrary string, and can itself be another circuit or program. We think of prm as being "hard wired" in the description of the corresponding circuit/program. The input to a circuit/program is specified inside parenthesis, so $C[\mathsf{prm}](x)$ describes the computation of the circuit $C[\mathsf{prm}]$ (whose definition depends on prm) on the input x.

3 Reusable GRAM without Persistent Memory

3.1 Definitions

We begin by defining non-reusable (one-time) and reusable garbled RAM. The syntax of the scheme is the same in both cases, and the difference is only in the security requirements.

Definition 3.1 (GRAM without persistent memory)**.** *A* garbled RAM *scheme without persistent memory consists of procedures* $\mathsf{GR} = (\mathsf{GR.prog}, \mathsf{GR.inp}, \mathsf{GR.eval})$*:*

- $(\tilde{P}, s) \leftarrow \mathsf{GR.prog}(1^\lambda, P, (n, m, t))$ *: Gets a RAM program P, and bounds on the program's: input size n, output size m, and run-time t (say that all bounds encoded in binary). Outputs a garbled program $\tilde{P}$ and a garbling key s.*
- $\tilde{x} \leftarrow \mathsf{GR.inp}(x, s, (n, m, t))$ *: Takes as input an n-bit value x, the garbling key s the same bounds (n, m, t). It outputs the garbled input $\tilde{x}$.*
- $y = \mathsf{GR.eval}(\tilde{P}, \tilde{x})$*: This is a RAM program that takes $(\tilde{P}, \tilde{x})$ as input and computes the output y.*

We require that for any program P with parameters (n, m, t), any input $x \in \{0,1\}^n$ if $\tilde{P}, \tilde{x}$ are created as described, then $\mathsf{GR.eval}(\tilde{P}, \tilde{x}) = P(x)$ with probability 1.

Definition 3.2 (GRAM Security)**.** *Let* GR *be a garbled RAM scheme as above.*

- GR *has reusable security if there exists a PPT simulator* Sim *such that, for all RAM programs P with polynomial parameters (n, m, t) and all polynomial-length input-vectors $(x_1, \ldots, x_q)$, the following distributions are computationally indistinguishable:*

$$(\tilde{P}, \tilde{x}_1, \ldots, \tilde{x}_q) \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\lambda, (n, m, t), y_1, \ldots, y_q)$$

where $(\tilde{P}, s) \leftarrow \mathsf{GR.prog}(1^\lambda, P, (n, m, t)), \tilde{x}_i \leftarrow \mathsf{GR.inp}(x_i, s, (n, m, t))$ and $y_i = P(x_i)$. The simulator is required to run in time $\mathsf{poly}(\lambda, n, m, t, q)$.

- GR *has* security with public input garbling *if there exists a PPT simulator* Sim' *such that the above security holds even when including the input-garbling key* s *in the left-hand distribution,*

$$(\tilde{P}, s, \tilde{x}_1, \ldots, \tilde{x}_q) \overset{\text{comp}}{\approx} \mathsf{Sim}'(1^\lambda, (n, m, t), y_1, \ldots, y_q).$$

- GR *has* one-time *security (resp. one-time security with public input garbling) if the above only holds for* $q = 1$.

Efficiency. We require that: $\mathsf{GR.prog}$ runs in time $\widetilde{O}(|P| + n + m + t) \cdot \mathsf{poly}(\lambda)$ and can be though of as a one-time preprocessing where the client has to work as hard as the program execution. $\mathsf{GR.inp}$ runs in time $\widetilde{O}(m + n) \cdot \mathsf{poly}(\lambda)$, and therefore is asymptotically efficient even as t becomes large. $\mathsf{GR.eval}$ runs in time $\widetilde{O}(|P| + n + m + t) \cdot \mathsf{poly}(\lambda)$ on a RAM, and therefore is only linear in the original program's running time t. In addition, we require that $\mathsf{GR.prog}, \mathsf{GR.inp}$ can be expressed as *circuits* with the above bounds denoting their circuit size. On the other hand, $\mathsf{GR.eval}$ is crucially expressed as a RAM program.

Remark on Program Privacy. Note that our definition does not explicitly consider *program privacy* and we assume that the code of the program P is public. This can be fixed via standard transformations, see Section 6.

We rely on prior constructions of one-time garbled RAM schemes [LO13a, GHL+14].[7]

Theorem 3.3 ([LO13a, GHL+14])**.** *Assuming the existence of selectively-secure identity-based encryption (IBE), there exist garbled RAM schemes with* one-time *security satisfying the above efficiency requirements.*

Garbled Circuits. As a useful tool, we will rely on the notion of reusable garbled circuits and we define the syntax of such schemes as follows.

Definition 3.4 (Garbled Circuits)**.** A *garbled circuit* scheme consists of three procedures, $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$:

- $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C)$: Gets an input circuit C and outputs a garbled circuit $\tilde{C}$ and key s.
- $\tilde{x} \leftarrow \mathsf{GC.inp}(x, s)$: Gets an input x and the same key s. Outputs the garbled input $\tilde{x}$.
- $y \leftarrow \mathsf{GC.eval}(\tilde{C}, \tilde{x})$: Gets a garbled circuit $\tilde{C}$ and matching input $\tilde{x}$, and computes the output.

We require that for any circuit C, input x, setting $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C)$ and $\tilde{x} \leftarrow \mathsf{GC.inp}(x, s)$ we get $\mathsf{GC.eval}(\tilde{C}, \tilde{x}) = C(x)$.

We defer discussion of the security properties that we need until later.

Output-size independent efficiency. Our main efficiency requirement is that $\mathsf{GC.inp}$ works in time $|x| \cdot \mathrm{poly}(\lambda)$, in particular its running time can *only* depend on the input size and not on the circuit size or even the output size. We call this requirement *output-size independent efficiency.* In addition, we require that $\mathsf{GC.circ}$ works in time $|C| \cdot \mathrm{poly}(\lambda)$ and that $\mathsf{GC.eval}$ works in time $\widetilde{O}(|\tilde{C}| + |\tilde{x}|) \cdot \mathsf{poly}(\lambda) = \widetilde{O}(|C| + |x|) \cdot \mathsf{poly}(\lambda)$.

[7] The syntax of [GHL+14] includes a separate procedure $\mathsf{GR.data}$ to garble memory, but for now we can think of this as part of the $\mathsf{GR.prog}$ procedure. We mention that similar result with slightly worse efficiency can be achieved under one-way functions.

3.2 Construction of Reusable GRAM

Overview. We construct a reusable garbled-RAM scheme by combining reusable garbled circuit with a one-time garbled RAM scheme. Let $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ be a reusable garbled circuit scheme whose required security properties we specify later and $\mathsf{GR1} = (\mathsf{GR1.prog}, \mathsf{GR1.inp}, \mathsf{GR1.eval})$ be a one-time garbled-RAM scheme. Recall our first approach from the introduction, which was to consider the circuit $C[P](r, x)$ that has P hard-wired in and gets as input randomness r and input x. The circuit $C[P](r, x)$ runs $(\tilde{P}_{one}, s) \leftarrow \mathsf{GR1.prog}(1^\lambda, P, (\cdots))$ and $\tilde{x}_{one} \leftarrow \mathsf{GR1.inp}(x, s, (\cdots))$, using r as randomness, and outputs $(\tilde{P}_{one}, \tilde{x}_{one})$. Our hope was to create a reusable garbled circuit $\tilde{C} \leftarrow \mathsf{GC.circ}(C[P])$ as our reusable garbled RAM program.

Observe that the circuit $C[P]$ from above has short input and very long output, related to the running time of P. Unfortunately, the construction of reusable garbled circuits of Goldwasser et al. [GKP+13b], requires that the size of the *garbled input* to the circuit always exceeds the size of the circuit's *output*, even if the size of the *actual input* of the circuit is small. In particular, they have output-size dependence. In Appendix C, we show that *any* reusable circuit garbling scheme with simulation-based security must have output-size dependence. In our context, that would mean that garbling an input to the program takes as much time as evaluating a program and therefore would not be useful in the context of outsourcing.

We fix the problem above by tweaking the above construction in a way that allows us to reduce the security requirement on the reusable garbled circuit to something weaker than simulation security, while still achieving simulation security for our final construction. In particular, we first present our modified construction of reusable garbled RAM from reusable garbled circuits, then we present a new notion of security for reusable garbled circuits that we call "distributional indistinguishability", and show that is suffices to make our construction secure (Section 3.3) and finally, we show how to instantiate this new notion of reusable garbled circuits (Section 3.4) while achieving output-size independent efficiency.

The main modification that we make to the first-attempt construction from above is to first transform a program P with run-time t into a modified program P^+ that we call the *real-or-dummy* program. In addition to the input x, the new program P^+ takes also an alleged output y and a flag ψ. If $\psi = 1$ (real) then $P^+(x, \psi, y)$ simply executes $P(x)$, ignoring y. If If $\psi = 0$ (dummy), on the other hand, then P^+ simply executes t dummy steps and outputs y, ignoring x.

Just as before, we consider the circuit $C[P^+](r, (x, \psi, y))$ that outputs a one-time garbled program $\tilde{P}$ garbling P^+ and garbled input $\tilde{x}$ garbling (x, ψ, y) using r as randomness. We then construct a reusable garbled circuit $\tilde{C}$ garbling $C[P^+]$ as the reusable garbled RAM program. Intuitively, the simulator of the reusable garbled RAM will simply provide a garbling of a "dummy input" consisting of $(r, 0^n, \psi = 0, y)$ instead of the "real" input $(r, x, \psi = 1, 0^m)$. Proving security of the new construction boils down to proving that, given $\tilde{C}$, one cannot distinguish many garbling of real inputs vs. dummy inputs. Notice that the outputs $\tilde{P}, \tilde{x}$ derived from real-inputs vs. dummy-inputs look indistinguishable by the security of the one-time garbled RAM. Therefore, we reduce simulation-based security of the full scheme to showing a new type of "distributional indistinguishability" for reusable garbled circuits, where it should be hard to distinguish garbled inputs from two different distributions (e.g., read or dummy) that produce indistinguishable outputs. This idea is similar in spirit to one used by De Caro et al. [CIJ+13] to convert indistinguishability-based security to simulation-based security for functional encryption. However, our notion of "distributional indistinguishability" is new.

The real-or-dummy program $\mathbf{P^+}$. In more detail, for a RAM program P with input-size n, output-size m and running-time bound t, let P^+ be a RAM program that gets as input (x, ψ, y) with $|x| = n$, $|\psi| = 1$ and $|y| = m$. If $\psi = 1$ then $P^+(x, \psi, y)$ outputs $P(x)$, and if $\psi = 0$ then it

outputs y after t steps. Note that the complexity of P^+ is essentially the same as P, except that it has input of size $n+m+1$ rather than just n.

The program-garbling circuit. A central component of our construction is a circuit that runs the program- and input-garbling routines of the underlying one-time GRAM scheme. For a RAM program P with input-size n, output-size m and running-time bound t, and for security parameter λ, define $C[P,n,m,t,\lambda]$ as the following circuit with $n+m+2\lambda+1$ input bits:[8]

$\underline{\mathbf{C}[\mathbf{P},\mathbf{n},\mathbf{m},\mathbf{t},\lambda](r,x,\psi,y)}$: // $r=(r_1,r_2)\in\{0,1\}^{2\lambda}$, $x\in\{0,1\}^n$, $\psi\in\{0,1\}$, $y\in\{0,1\}^m$

1. Run $(\tilde{P},s)\leftarrow \mathsf{GR1.prog}(1^\lambda,P^+,(n,m,t);r_1)$, $\tilde{x}\leftarrow \mathsf{GR1.inp}((x,\psi,y),s,(n,m,t);r_2)$,
2. Output $(\tilde{P},\tilde{x})$.

Recall that for the program P^+ as above, the circuit-size of $\mathsf{GR1.prog}$ is $\widetilde{O}(|P|+n+m+t)\cdot\mathrm{poly}(\lambda)$ and the circuit-size of $\mathsf{GR1.inp}$ is $O(n+m)\cdot\mathrm{poly}(\lambda)$. Hence the size of $C[P,n,m,t,\lambda]$ (as well as the time that it takes to generate its description) can be bounded by $\widetilde{O}(|P|+n+m+t)\cdot\mathrm{poly}(\lambda)$.

GR: Reusable Garbled RAM Construction. We describe our reusable GRAM construction $\mathsf{GR}=(\mathsf{GR.prog},\mathsf{GR.inp},\mathsf{GR.eval})$ in the following figure.

$\underline{\mathsf{GR.prog}(1^\lambda,P,(n,m,t))}$:

1. Construct the circuit $C[P,n,m,t,\lambda]$ as shown above.
2. Output a garbling of this circuit $(\tilde{C},s)\leftarrow\mathsf{GC.circ}(1^\lambda,C[P,n,m,t,\lambda])$.

$\underline{\mathsf{GR.inp}(x,s,(n,m,t))}$:

1. Choose a random $r\leftarrow\{0,1\}^{2\lambda}$, set $\psi=1,y=0^m$, and $w=(r,x,\psi,y)$,
2. Garble the input to $C[P\ldots]$, outputting $\tilde{w}\leftarrow\mathsf{GC.inp}(w,s)$.

$\underline{\mathsf{GR.eval}(\tilde{C},\tilde{w})}$:

1. Evaluate the garbled circuit, $(\tilde{P},\tilde{x})\leftarrow\mathsf{GC.eval}(\tilde{C},\tilde{w})$, // $=C[P,n,m,t,\lambda](s,x,\psi,y)$
2. Evaluate the 1-time GRAM and output $y\leftarrow\mathsf{GR1.eval}(\tilde{P},\tilde{x})$. // $=P(x)$

Functionality and complexity. The correctness of this scheme can be verified by inspection. As for its complexity, since the size of $C[P,n,m,t,\lambda]$ and the time to construct it are bounded by $\widetilde{O}(|P|+n+m+t)\cdot\mathrm{poly}(\lambda)$, the overall complexity of $\mathsf{GR.prog}$ is also $\widetilde{O}(|P|+n+m+t)\cdot\mathrm{poly}(\lambda)$. The input to $C[P,n,m,t,\lambda]$ has length $O(n+m+\lambda)$, and therefore the time to garble that input (which is the complexity of $\mathsf{GR.inp}$) is bounded by $O(n+m)\cdot\mathrm{poly}(\lambda)$. [9] Finally, the time to evaluate the garbled circuit is polynomial in its size, hence the first step of $\mathsf{GR.eval}$ has complexity $\widetilde{O}(|P|+n+m+t)\cdot\mathrm{poly}(\lambda)$. Also, the time to evaluate a garbling of P^+ is essentially the running time $\mathsf{GR1.eval}(\tilde{P},\tilde{x})$ is essentially that of P^+, so also the second step has complexity bounded by $\widetilde{O}(|P|+n+m+t)\cdot\mathrm{poly}(\lambda)$ as well, and so this term bounds the overall complexity of $\mathsf{GR.eval}$.

[8] For simplicity, we assume that $\mathsf{GR1.prog}$, $\mathsf{GR1.inp}$ uses exactly λ bits of randomness each. This can always be made the case by using a pseudorandom generator.

[9] Note that the complexity of input garbling depends on *both the input and the output size of P*, meaning that our overall construction has output-size dependence. This is necessary for reusable simulation-security (see Appendix C).

3.3 Simulation Security From Distributional Indistinguishability

A crucial observation is that we can prove simulation-security for the above reusable GRAM construction GR using a new notion of "distributional indistinguishability" security for the underlying garbled circuit scheme and the usual simulation security for the underlying one-time GRAM scheme. "Distributional indistinguishability' says that one cannot distinguish garbled inputs from any two sets of independent distributions that produce individually indistinguishable outputs.

Definition 3.5. *Let* $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ *be a garbled circuit scheme. We say that* GC *provides* distributional indistinguishability *if for every circuit ensemble* $C = \{C_\lambda\}$*, every polynomial* $q = q(\lambda)$*, and every* $2q$ *polynomial-time samplable distributions* $D_1, \ldots, D_q$ *and* $D'_1, \ldots, D'_q$*, if for all* $j \in [q]$ *it holds that* $C(w_j) \overset{\text{comp}}{\approx} C(w'_j)$ *where* $w_j \leftarrow D_j(1^\lambda), w'_j \leftarrow D'_j(1^\lambda)$ *then it also holds that*

$$\langle \tilde{C}, \tilde{w}_1, \ldots \tilde{w}_q \rangle \overset{\text{comp}}{\approx} \langle \tilde{C}, \tilde{w}'_1, \ldots \tilde{w}'_q \rangle$$

where $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C_\lambda)$, $w_i \leftarrow D_i(1^\lambda)$, $w'_i \leftarrow D'_i(1^\lambda)$, $\tilde{w}_i \leftarrow \mathsf{GC.inp}(w_i, s)$, $\tilde{w}'_i \leftarrow \mathsf{GC.inp}(w'_i, s)$.

We say that the scheme has security with public input garbling *if the above holds when we include the garbling key* s *in the two distributions on the bottom.*

We remark that this notion is clearly implied by simulation security for reusable garbled circuits, but simulation security of reusable garbled circuits requires "output-size dependence" where the size of the garbled input must exceed that of the circuit's output (see Appendix C). Furthermore, we remark that for any scheme with public input garbling, distributional indistinguishability for $q = 1$ implies security for arbitrary q by a simple hybrid argument. Interestingly, this does not hold for simulation security where it may be possible to simulate $q = 1$ inputs but not possible to simulate for a larger q, even if the scheme has public input garbling.

In Section 3.4 we show how to construct a reusable circuit-garbling scheme with output-size independence satisfying this definition using indistinguishability obfuscation.

Theorem 3.6. *If* $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ *is a reusable garbled-circuit scheme satisfying distributional indistinguishability and output-independent efficiency, and* $\mathsf{GR1} = (\mathsf{GR1.prog}, \mathsf{GR1.inp}, \mathsf{GR1.eval})$ *is a one-time garbled-RAM scheme, then the scheme* GR *from above is a reusable garbled-RAM scheme satisfying simulation security. Furthermore, if* GC *has security with public input garbling, then so does* GR*.*

Proof. The simulator was sketched above: On input $(1^\lambda, (n, m, t), P, y_1, \ldots, y_q)$ with $|y_i| = m$ for all i, the simulator $\mathsf{GR.sim}$ begins just as the garbling procedure of the actual scheme, namely by constructing the circuit $C[P, n, m, t, \lambda]$ and applying to it the circuit-garbling procedure to get $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C[P, n, m, t, \lambda])$. Next, for every y_i the simulator chooses a uniformly random $r_i \in \{0,1\}^{2\lambda}$, sets $w'_i = (r_i, 0, \psi = 0, y_i)$ and $\tilde{w}'_i \leftarrow \mathsf{GC.inp}(w_i, s)$. The output of the simulator $\mathsf{GR.sim}$ consists of $(\tilde{C}, \tilde{w}'_1, \ldots, \tilde{w}'_q)$.

We next prove indistinguishability between the real and simulated outputs. Fix the program P and inputs $x_1, \ldots, x_q$ for P, and denote $y_i = P(x_i)$ for all i. Let $w_i = (r_i, x_i, \psi = 1, 0^m)$ where $r_i \in \{0,1\}^{2\lambda}$. We first argue that the output distributions on inputs w_i and w'_i are indistinguishable.

Claim 3.7. *Denote* $C = C[P, n, m, t, \lambda]$*. If the scheme* $\mathsf{GR1} = (\mathsf{GR1.prog}, \mathsf{GR1.inp}, \mathsf{GR1.eval})$ *satisfies one-time GRAM security then for every* $x \in \{0,1\}^n$ *and* $y = P(x)$ *we have*

$$\{C(w_i)\} \overset{\text{comp}}{\approx} \{C(w'_i)\},$$

where w_i, w'_i *are chosen as described above.*

Proof. Note that $C(w_i) = (\tilde{P}_i, \tilde{x}_i)$ and $C(w'_i) = (\tilde{P}_i, \tilde{x}'_i)$ where $\tilde{P}_i$ is a garbled version of the program P^+, $\tilde{x}_i$ is a garble version of the input $(x_i, 1, 0^m)$ and $\tilde{x}'_i$ is a garbled version of the input $(0^n, 0, y_i)$. The one-time simulation-security of the underlying $\mathsf{GR1}$ scheme implies that:

$$(\tilde{P}_i, \tilde{x}_i) \overset{\text{comp}}{\approx} \mathsf{GR1.Sim}(1^\lambda, (n, m, t), y_i) \overset{\text{comp}}{\approx} (\tilde{P}_i, \tilde{x}'_i).$$

since $y_i = P^+((x_i, 1, 0^m)) = P^+((0^n, 0, y_i))$. This proves the claim. □

Claim 3.7 implies that the distributions of the w_i, w'_i's satisfy the condition of Definition 3.5, and by the distributional indistinguishability of GC we conclude that also

$$\langle \tilde{C}, \tilde{w_1}, \ldots \tilde{w_q} \rangle \overset{\text{comp}}{\approx} \langle \tilde{C}, \tilde{w'_1}, \ldots \tilde{w'_q} \rangle.$$

This completes the proof, since these are exactly the output distributions of the scheme GR and its simulator $\mathsf{GR.sim}$. □

3.4 Achieving Distributional Indistinguishability

We now construct reusable garbled circuits with "distributional indistinguishability" and "output-size independent efficiency". Furthermore, our construction has public input-garbling. The construction is based on "indistinguishability obfuscation" (see Appendix A.1) and a NIZK which is "statistically simulation sound" (see Appendix A.2). It is inspired by the construction of functional encryption from indistinguishability obfuscation of [GGH+13].

Construction. Let $\mathcal{O}$ be an obfuscation scheme, let $\mathcal{PKE} = (\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{Decrypt})$ be a public key encryption scheme, and let $\Pi = (K, P, V)$ be a NIZK scheme with statistical simulation soundness. Let L_{EQ} be the NP language defined as

$$L_{EQ} = \{ \ (\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \ : \ \exists\, m, r_1, r_2, \ \ c_1 = \mathsf{Encrypt}(\mathsf{pk}_1, m; r_1) \wedge c_2 = \mathsf{Encrypt}(\mathsf{pk}_2, m; r_2) \} .$$

For any circuit $C : \{0,1\}^n \to \{0,1\}^m$ define the circuit $C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, b, \mathsf{sk}, u, v]$ where: σ is a CRS for the NIZK, $\mathsf{pk}_1, \mathsf{pk}_2$ are encryption keys, $b \in \{1,2\}$ is an index, sk is the decryption key for pk_b, u is of size $|(c_1, c_2, \pi)|$ where c_1, c_2 are ciphertexts of n-bit messages and π is a NIZK for L_{EQ}, and $v \in \{0,1\}^m$.

$C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, b, \mathsf{sk}, u, v](c_1, c_2, \pi)$:
1. If $u = (c_1, c_2, \pi)$ output v.
2. Verify that π is a proof of $(\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \in L_{EQ}$ by running $V(\sigma, (\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2), \pi)$. If this rejects, output $\perp$.
3. Compute $x = \mathsf{Decrypt}(\mathsf{sk}, c_b)$. Output $C(x)$.

We define the circuit garbling scheme $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$, which has public input garbling, as follows:

- $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C)$: Generate $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{Setup}(1^\lambda)$, $\sigma \leftarrow K(1^\lambda)$. Construct the circuit $C^* := C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, 1, \mathsf{sk}_1, u = \perp, v = \perp]$ from C as shown. Output $\tilde{C} \leftarrow \mathcal{O}(C^*)$ and $s := (\sigma, \mathsf{pk}_1, \mathsf{pk}_2)$.
- $\tilde{x} \leftarrow \mathsf{GC.inp}(x, s)$: output $\tilde{x} := (c_1, c_2, \pi)$, where $c_1 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_1, x; r_1)$, $c_2 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_2, x; r_2)$ and $\pi \leftarrow P(\sigma, (\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2), (r_1, r_2))$ is an NIZK that $(\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \in L_{EQ}$.
- $\mathsf{GC.eval}(\tilde{C}, \tilde{x})$: Interpret $\tilde{C}$ as an obfuscated circuit and output $\tilde{C}(\tilde{x})$.

Theorem 3.8. *If $\mathcal{O}$ is an indistinguishability obfuscator, Π is a statistical-simulation-sound (SSS) NIZK, and $\mathcal{PKE}$ is a semantically secure encryption scheme, then the above construction GC is a* reusable garbled circuit with distributional indistinguishability *and* public input garbling*. In particular, such schemes exist given the existence of indistinguishability obfuscation and one-way functions.*

Proof. Let $C = \{C_\lambda\}$ be a circuit (ensemble) and $D_1, \ldots, D_q$ and $D'_1, \ldots, D'_q$ be efficiently samplable input distributions (ensembles) such that, for all $j = 1, \ldots, q$ it holds that

$$\{C(x_j) : x_j \leftarrow D_j(1^\lambda)\} \overset{\text{comp}}{\approx} \{C(x'_j) : x'_j \leftarrow D'_j(1^\lambda)\}.$$

We will show that:

$$\begin{aligned} &\left\{\langle s, \tilde{C}, \tilde{x}_1, \ldots \tilde{x}_q\rangle : x_i \leftarrow D_i(1^\lambda),\ (\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C),\ \tilde{x}_i \leftarrow \mathsf{GC.inp}(x_i, s)\right\} \\ \overset{\text{comp}}{\approx}\ &\left\{\langle s, \tilde{C}, \tilde{x}'_1, \ldots \tilde{x}'_q\rangle : x'_i \leftarrow D'_i(1^\lambda),\ (\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C),\ \tilde{x}'_i \leftarrow \mathsf{GC.inp}(x'_i, s)\right\} \end{aligned}$$

Since we consider public input garbling (by including s in the distributions), we can rely on a simple hybrid argument to show that the above holds as long as for each $i \in [q]$:

$$\langle s, \tilde{C}, \tilde{x}_i\rangle \overset{\text{comp}}{\approx} \langle s, \tilde{C}, \tilde{x}'_i\rangle \tag{1}$$

In particular, given s, we can efficiently sample the values $x_j \leftarrow D_j(1^\lambda), \tilde{x}_j \leftarrow \mathsf{GC.inp}(x_j, s)$ as well as $x'_j \leftarrow D'_j(1^\lambda), \tilde{x}'_j \leftarrow \mathsf{GC.inp}(x'_j, s)$ for all $j \neq i$ ourselves.

To show (1), we define the following hybrid distributions:

- Hyb_0: This is the distribution $\langle s, \tilde{C}, \tilde{x}_i\rangle$. Recall that $s = (\sigma, \mathsf{pk}_1, \mathsf{pk}_2)$, $\tilde{C} = \mathcal{O}(C^*)$, where $C^* = C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, 1, \mathsf{sk}_1, u = \bot, v = \bot])$, and $\tilde{x}_i = (c_1, c_2, \pi)$.

- Hyb_1: In this hybrid, we switch the real proof to a simulated one for the statement $\mathsf{st} = (\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2)$. In particular, we now choose $(\sigma, \tau) \leftarrow S_1(1^\lambda, \mathsf{st})$ to be a simulated CRS and $\pi \leftarrow S_2(\sigma, \tau, \mathsf{st})$ where (S_1, S_2) are the simulators of the NIZK scheme. We claim:

  $$\mathsf{Hyb}_0 \overset{\text{comp}}{\approx} \mathsf{Hyb}_1$$

  This follows directly from the computational zero-knowledge property of the NIZK Π.

- Hyb_2: In this hybrid, we encrypt $c_2 \leftarrow \mathsf{Enc}(\mathsf{pk}_2, \bar{0})$ in the second ciphertext, where $\bar{0}$ is the same length as x_i. The first ciphertext c_1 still encrypts $x_i \leftarrow D_i(1^\lambda)$. We claim:

  $$\mathsf{Hyb}_1 \overset{\text{comp}}{\approx} \mathsf{Hyb}_2$$

  This follows directly from the semantic security of the encryption scheme. Notice that sk_2 does not appear anywhere in either hybrid.

- Hyb_3: In this hybrid, we switch the circuit being obfuscated from $C_1^* = C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, 1, \mathsf{sk}_1, u = \bot, v = \bot]$ to $C_2^* = C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, 2, \mathsf{sk}_2, u = (c_1, c_2, \pi), v = y_i]$ where $y_i = C(x_i)$. Notice C_2^* now uses sk_2 to decrypt but has the values u, v hard-coded. We claim that C_1^*, C_2^* are functionally equivalent: for any input $u' = (c'_1, c'_2, \pi')$ such that $u' \neq u$ and π' verifies, the ciphertexts (c'_1, c'_2) must encrypt the same message (by the statistical-simulation-soundness of the NIZK) and hence $C_1^*(u') = C_2^*(u')$. On the other hand, for input u, we have $C_1^*(u) = C_2^*(u) = y_i$. We claim:

  $$\mathsf{Hyb}_2 \overset{\text{comp}}{\approx} \mathsf{Hyb}_3$$

  This follows directly from the indistinguishability obfuscation property, by noting the C_1^* and C_2^* are functionally equivalent.

- Hyb_4: In this hybrid, we also encrypt $c_1 \leftarrow \mathsf{Enc}(\mathsf{pk}_1, \bar{0})$ in the first ciphertext, where $\bar{0}$ is the same length as x_i. We claim:

$$\mathsf{Hyb}_3 \overset{\text{comp}}{\approx} \mathsf{Hyb}_4$$

  This follows directly from the semantic security of the encryption scheme. Notice that sk_1 does not appear anywhere in either hybrid.

- Hyb_5: In this hybrid, we switch the hard-coded value in $C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, 2, \mathsf{sk}_2, u, v]$ from $v = y_i$ to $v = y'_i = C(x'_i)$ where $x'_i \leftarrow D'_i(1^\lambda)$. We claim:

$$\mathsf{Hyb}_4 \overset{\text{comp}}{\approx} \mathsf{Hyb}_5$$

  This follows from the condition that $C(x_i) \overset{\text{comp}}{\approx} C(x'_i)$. Notice that no other information about x_i, x'_i other than $y_i = C(x_i), y'_i = C(x'_i)$ appears in either hybrid.

- Hyb_6 to Hyb_{10}: These are the same as Hyb_4 through Hyb_0 but using (x'_i, y'_i) in place of (x_i, y_i). Notice that Hyb_{10} is just the distribution $\langle s, \tilde{C}, \tilde{x}'_i \rangle$. By symmetry, we get:

$$\mathsf{Hyb}_5 \overset{\text{comp}}{\approx} \mathsf{Hyb}_{10}$$

Combining the above, we get $\mathsf{Hyb}_0 \overset{\text{comp}}{\approx} \mathsf{Hyb}_{10}$, which proves equation (1) and the theorem follows. □

Summary. Combining the above with Theorem 3.3 and 3.6, and the facts that indistinguishability obfuscation + one-way function implies selectively secure functional encryption which implies IBE [GGH+13], and that statistically simulation sound NIZK can be constructed from statistically sounds NIZK [GGH+13], we get the following corollary.

Corollary 3.9. *If indistinguishability obfuscation, one-way functions, and statistically sounds NIZKs exist, then there exists a reusable garbled-RAM scheme without persistent memory satisfying Definition 3.2. Furthermore, it supports public input garbling.*

4 Reusable Garbled RAM with Persistent Memory

We now move to consider the harder setting with persistent memory. The construction is very similar to the one above, in particular here too we construct a reusable garbled-RAM scheme by combining a reusable garbled circuit with a one-time garbled RAM scheme. The main differences are that the one-time GRAM that we use has persistent memory, and the garbled circuit satisfies a stronger notion of security.

4.1 Definitions

A GRAM scheme with persistent memory has an additional procedure $\mathsf{GR.data}$ used to garbled the initial memory data D to a garbled data $\tilde{D}$. We envision the case where the user has a program P and wants to run many executions of this program with different "short-inputs" x_i where each execution $P(x_i)$ can read and write to the persistent memory, and the changes persist for future executions. This means that the order of executions is important and does not commute. In particular, we need to ensure that the server only learns the outputs of the executions when performed in the correct order and cannot (e.g.,) reorder the executions or roll-back the changes made by one execution when performing the next execution.

In order to do this, we need to incorporate the index i of the execution into the program/input garbling procedures. In the case of reusable schemes the user garbled the program only once to get the garbled version $\tilde{P}$, and then can garble many different inputs $\tilde{x}_i$, so in this case only the *input* garbling is given the index i. For one-time scheme the user has to garble the program P afresh for each new execution to get a garbled program $\tilde{P}_i$ and garbled input $\tilde{x}_i$, so in this case both the *program* and *input* garbling procedures are given the index i.

Definition 4.1 (GRAM with persistent memory)**.** *A* garbled RAM *(GRAM) scheme* with persistent memory *consists of four procedures:* $\mathsf{GR} = (\mathsf{GR.data}, \mathsf{GR.prog}, \mathsf{GR.inp}, \mathsf{GR.eval})$*:*

- $(\tilde{D}, k) \leftarrow \mathsf{GR.data}(1^\lambda, D)$ *: Gets data* $D \in \{0,1\}^N$ *and outputs the garbled data* $\tilde{D}$ *and a data-key* k*.*
- $(\tilde{P}, s) \leftarrow \mathsf{GR.prog}(P, k, (N, n, m, t), [i])$ *: Gets a RAM program* P*, data-key* k*, and bounds on the memory size* N*, input size* n*, output size* m*, and run-time* t *(say that all bounds encoded in binary). For one-time scheme, the procedure is also given an index* i *indicating the order in which this program is to be executed. Outputs a garbled program* $\tilde{P}$ *and program-key* s*.*
- $\tilde{x} \leftarrow \mathsf{GR.inp}(x, k, s, (N, n, m, t), i)$*: Takes as input an* n*-bit value* x*, the keys* k, s*, and an index* i *for the order of the execution. It outputs the garbled input* $\tilde{x}$*.*
- $y = \mathsf{GR.eval}^{\tilde{D}}(\tilde{P}, \tilde{x})$*: This is a RAM program that takes* $(\tilde{P}, \tilde{x})$ *as input, has memory* $\tilde{D}$*, and computes the output* y*. The program updates the memory contents of* $\tilde{D}$ *during its execution and these changes persist for the following execution.*

Correctness. Consider initial memory data $D \in \{0,1\}^N$, program P with bounds (N, n, m, t) and a sequence of inputs $x_i \in \{0,1\}^n$ for $i = 1, \ldots, q$. Let $y_i = P^D(x_i)$ where the executions are performed in the correct order starting with $i = 1$ and the memory data D is updated with each execution. We define one-time and reusable correctness separately.

One-time: Consider running $(\tilde{D}, k) \leftarrow \mathsf{GR.data}(1^\lambda, D)$, $(\tilde{P}_i, s_i) \leftarrow \mathsf{GR.prog}(P, k, (N, n, m, t), i)$, $\tilde{x}_i \leftarrow \mathsf{GC.inp}(x_i, k, s_i)$, $y'_i \leftarrow \mathsf{GC.eval}^{\tilde{D}}(\tilde{P}_i, \tilde{x}_i)$ where the evaluations are executed in the correct order and the garbled memory $\tilde{D}$ is updated with each evaluation. We require that $y'_i = y_i$ for all $i \in [q]$ with probability 1.

Reusable: Consider running $(\tilde{D}, k) \leftarrow \mathsf{GR.data}(1^\lambda, D)$, $(\tilde{P}, s) \leftarrow \mathsf{GR.prog}(P, k, (N, n, m, t))$, $\tilde{x}_i \leftarrow \mathsf{GC.inp}(x_i, k, s, i)$, $y'_i \leftarrow \mathsf{GC.eval}^{\tilde{D}}(\tilde{P}, \tilde{x}_i)$ where the evaluations are executed in the correct order and the garbled memory $\tilde{D}$ is updated with each evaluation. We require that $y'_i = y_i$ for all $i \in [q]$ with probability 1.

Efficiency. We require that data-garbling $\mathsf{GR.data}$ runs in time $N \cdot \mathrm{poly}(\lambda)$, program-garbling $\mathsf{GR.prog}$ run in time $\widetilde{O}(|P|+n+m+t) \cdot \mathrm{poly}(\lambda, \log N)$, input-garbling $\mathsf{GR.inp}$ runs in time $\widetilde{O}(m+n) \cdot \mathrm{poly}(\lambda)$, and the execution time of $\mathsf{GR.eval}$ must be bounded by $\widetilde{O}(|P|+n+m+t) \cdot \mathrm{poly}(\lambda, \log N)$. Furthermore, we require that $\mathsf{GR.prog}$ and $\mathsf{GR.inp}$ can be expressed as circuits with the above efficiency denoting their size.

Security. We require the usual simulation-based security from our garbled-RAM constructions. To simplify notation, we define a polynomial size *input specification* $\mathsf{in} = ((N, n, m, t), D, P, \langle x_i \rangle_{i \in [q]})$ as consisting of polynomial bounds (N, n, m, t), initial memory data $D \in \{0,1\}^N$, program P, and a polynomial-size sequence of inputs $x_i \in \{0,1\}^n$ for $i = 1, \ldots, q$. We define the corresponding *output specification* $\mathsf{out} = ((N, n, m, t), P, \langle y_i \rangle_{i \in [q]})$ with $y_i = P^D(x_i)$ where the executions are performed in the correct order starting with $i = 1$ and the memory data D is updated with each

execution. Intuitively, the output specification is the only thing that the evaluator should learn from the garbled data/program/input.

Definition 4.2 (GRAM with Persistent Memory)**.** *Let* $\mathsf{GR} = (\mathsf{GR.data}, \mathsf{GR.prog}, \mathsf{GR.inp}, \mathsf{GR.eval})$ *be a garbled RAMs with persistent memory. We say* GR *has one-time/reusable security, if there exists a simulator* Sim *such that for any poly-size input specification* $\mathsf{in} = ((N, n, m, t), D, P, \langle x_i \rangle_{i\in[q]})$ *with corresponding output specification* $\mathsf{out} = ((N, n, m, t), P, \langle y_i \rangle_{i\in[q]})$ *we have*

$$\mathsf{Real}[\mathsf{in}, \lambda] \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\lambda, \mathsf{out}),$$

where $\mathsf{Real}[\mathsf{in}, \lambda]$ *is defined separately for one-time and reusable security as follows:*

One-Time. *Define* $\mathsf{Real}[\mathsf{in}, \lambda] = (\tilde{D}, \langle \tilde{P}_i, \tilde{x}_i \rangle_{i\in[q]})$ *where* $(\tilde{D}, k) \leftarrow \mathsf{GR.data}(1^\lambda, D)$*, and for* $i \in [q]$ $(\tilde{P}_i, s_i) \leftarrow \mathsf{GR.prog}(P, k, (N, n, m, t), i)$*,* $\tilde{x}_i \leftarrow \mathsf{GC.inp}(x_i, k, s_i)$*.*

Reusable. *Define* $\mathsf{Real}[\mathsf{in}, \lambda] = (\tilde{D}, \tilde{P}, \langle \tilde{x}_i \rangle_{i\in[q]})$ *where* $(\tilde{D}, k) \leftarrow \mathsf{GR.data}(1^\lambda, D)$*,* $(\tilde{P}, s) \leftarrow \mathsf{GR.prog}(P, k, (N, n, m, t))$*, and for* $i \in [q]$*:* $\tilde{x}_i \leftarrow \mathsf{GC.inp}(x_i, k, s_i, i)$*.*

We require that Sim *runs in time* $\mathsf{poly}(N, t, n, m, q, \lambda)$*.*

Remarks on Definition. For simplicity, we only consider the scenario involving a single program P. This is without loss of generality as P can be a *universal RAM* that executes code stored in memory at a location which is indicated as part of the short input x. This approach of storing code in memory most closely resembles computation in real life. We also do not explicitly consider *program privacy* and assume that the code of the program is public. (The approach of setting P to be a universal RAM and executing programs stored in memory also provides privacy for the code of the programs being executed, see Section 6.)

Note that garbled RAM with persistent memory *cannot* have public input garbling, as this allows the attacker to learn more information than allowed about the garbled database D by executing its own programs over it. Hence, the data key k is kept secret and the evaluator only learns the outputs of the specified program P with specified inputs x_i when executed in the correct order. This prevents the attacker from, e.g., learn the outputs of additional programs, roll back the changes to data made by one program and see how it affects the outputs of future programs, etc.

The works of [LO13a, GHL+14] provide constructions of one-time GRAM with persistent memory satisfying the above definition. The syntax of [GHL+14] is somewhat more complicated since it considers a scenario with many different programs P_i, but it is easy to see that it implies our simplified syntax/definition.

Theorem 4.3 ([LO13a, GHL+14])**.** *Assuming the existence of selectively-secure identity-based encryption (IBE), there exist garbled RAM schemes with persistent memory and* one-time *security satisfying the above efficiency requirements.*

4.2 Constructing Reusable GRAM with Persistent Memory

The construction of garbled RAM scheme with persistent memory from a one-time garbled RAM and a reusable garbled circuits is very similar to the case without persistent memory, we essentially just make the needed syntactic changes to accommodate the persistent memory.

Let $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ be a reusable garbled circuit scheme and $\mathsf{GR1} = (\mathsf{GR.data}, \mathsf{GR1.prog}, \mathsf{GR1.inp}, \mathsf{GR1.eval})$ be a one-time garbled-RAM scheme with persistent memory. For a program P, let P^+ be the "real-or-dummy" program defined previously. For a RAM program P, with memory-size N, input-size n, output-size m, running-time bound t, and security parameter λ, define $C[P, N, n, m, t, \lambda]$ as the following circuit with $n + m + 2\lambda + 1$ input bits:

$\underline{\mathbf{C}[\mathbf{P},\mathbf{N},\mathbf{n},\mathbf{m},\mathbf{t},\lambda](k,r=(r_1,r_2),x,\psi,y,i)}$: // $k,r_1,r_2 \in \{0,1\}^\lambda$
// $x \in \{0,1\}^n$, $\psi \in \{0,1\}$, $y \in \{0,1\}^m$

1. Run $(\tilde{P},s) \leftarrow \mathsf{GR1.prog}(P^+,k,(N,m+n+1,m,t),i;r_1)$,
   $\tilde{x} \leftarrow \mathsf{GR1.inp}((x,\psi,y),k,s;r_2)$,
2. Output $(\tilde{P},\tilde{x})$.

As before, the size of $C[P,n,m,t,\lambda]$ (as well as the time that it takes to generate its description) can be bounded by $\widetilde{O}(n+m+t)\cdot\mathrm{poly}(\lambda,\log N)$.

Reusable Garbled RAM with Persistent Memory. We describe the scheme GR in the following figure.

$\underline{\mathsf{GR.data}(1^\lambda,D)}$: Use the the underlying $\mathsf{GR1}$ and output: $(\tilde{D},k) \leftarrow \mathsf{GR1.data}(1^\lambda,D)$.

$\underline{\mathsf{GR.prog}(P,k,(N,n,m,t))}$:
1. Construct the circuit $C = C[P,N,n,m,t,\lambda]$,
2. Garble this circuit, outputting $(\tilde{C},s) \leftarrow \mathsf{GC.circ}(1^\lambda,C)$.

$\underline{\mathsf{GR.inp}(x,k,s,i)}$:
1. Choose a random $r \leftarrow \{0,1\}^{2\lambda}$, set $\psi = 1, y = 0^m$, and $w = (k,r,x,\psi,y,i)$,
2. Garble the input to $C[P\ldots]$, outputting $\tilde{w} \leftarrow \mathsf{GC.inp}(w,s,i)$.

$\underline{\mathsf{GR.eval}(\tilde{C},\tilde{w})}$:
1. Evaluate the garbled circuit, $(\tilde{P},\tilde{x}) \leftarrow \mathsf{GC.eval}(\tilde{C},\tilde{w})$, // $= C[P,N,n,m,t,\lambda](k,r,x,\psi,y,i)$
2. Evaluate the 1-time GRAM and output $y \leftarrow \mathsf{GR1.eval}(\tilde{P},\tilde{x})$. // $= P(x)$

Functionality and complexity. The correct functionality and complexity of GR are argued exactly as for the case without persistent memory. Here the complexity of $\mathsf{GR.data}$ is $N\cdot\mathrm{poly}(\lambda)$ as in the underlying one-time-GRAM scheme, the complexity of $\mathsf{GR.prog}$ is $\widetilde{O}(|P|+n+m+t)\cdot\mathrm{poly}(\lambda)$ (essentially due to the complexity of $C[P,\ldots]$), the complexity of $\mathsf{GR.inp}$ is bounded by $O(n+m)\cdot\mathrm{poly}(\lambda)$, and $\mathsf{GR.eval}$ has complexity $\widetilde{O}(|P|+n+m+t)\cdot\mathrm{poly}(\lambda)$.

Security. The proof of security too is very similar to the case of no persistent memory, except that we need a stronger variant of "distributional indistinguishability". In the previous case without persistent memory, each garbled input $w_i = (r_i, x_i, \psi = 1, 0^m)$ was chosen with its own fresh and independent randomness r_i and there was no relationship between different garbled inputs. In the case of persistent memory, the inputs $w_i = (k, r_i, x_i, \psi = 1, 0^m, i)$ all include the same "data key" k. In other words, the inputs have some common *correlated secret* and we cannot argue that they come from independent distributions D_i, D'_i. Therefore, we are forced to rely on a stronger notion of security of reusable garbled circuit than Definition 3.5, a notion we define below.

Definition 4.4 (Correlated Distributional Indistinguishability)**.** *Let* $\mathsf{GC} = (\mathsf{GC.circ},\mathsf{GC.inp},\mathsf{GC.eval})$ *be a garbled circuit scheme. We say that* GC *provides* correlated distributional indistinguishability *if for every circuit ensemble* $C = \{C_\lambda\}$ *and two PPT distribution samplers* D, D' *over vectors of inputs to* C*, if the two distributions satisfy*

$$\langle C_\lambda(w_1),\ldots,C_\lambda(w_q),\mathsf{aux}\rangle \overset{\text{comp}}{\approx} \langle C_\lambda(w'_1),\ldots,C_\lambda(w'_q),\mathsf{aux}'\rangle$$

where $(w_1, \ldots, w_q, \mathsf{aux}) \leftarrow D(1^\lambda)$, $(w'_1, \ldots, w'_q, \mathsf{aux}') \leftarrow D'(1^\lambda)$, *then also*

$$\langle \tilde{C}, \tilde{w}_1, \ldots \tilde{w}_q, \mathsf{aux} \rangle \overset{\text{comp}}{\approx} \langle \tilde{C}, \tilde{w}'_1, \ldots \tilde{w}'_q, \mathsf{aux}' \rangle$$

where $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C_\lambda)$ *and* $\tilde{w}_i \leftarrow \mathsf{GC.inp}(w_i, s)$, $\tilde{w}'_i \leftarrow \mathsf{GC.inp}(w'_i, s)$.

We note that this definition is still weaker than simulation security, but stronger than Definition 3.5 (which considers only w_i's that are sampled independently). The proof of the following theorem is similar to that of Theorem 3.6.

Theorem 4.5. *If* $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ *is a reusable garbled-circuit scheme with output-size independence satisfying correlated distributional indistinguishability and* $\mathsf{GR1} = (\mathsf{GR1.data}, \mathsf{GR1.prog}, \mathsf{GR1.inp}, \mathsf{GR1.eval})$ *is a garbled-RAM scheme with persistent memory satisfying one-time simulation security, then the scheme* GR *from above is a reusable garbled-RAM scheme with persistent memory satisfying simulation security (Definition 4.2).*

Proof. Let us fix some polynomial size input specification $\mathsf{in} = ((N, n, m, t), D, P, \langle x_i \rangle)$ with corresponding output specification $\mathsf{out} = ((N, n, m, t), P, \langle y_i \rangle)$ where $i = 1, \ldots, q$.

The simulator $\mathsf{GR.sim}$ gets input $(1^\lambda, \mathsf{out})$. It begins by garbling a dummy database $(\tilde{D}', k) \leftarrow \mathsf{GR.data}(1^\lambda, 0^N)$. It then constructs the circuit $C = C[P, N, n, m, t, \lambda]$ and applies the circuit-garbling procedure to get $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C)$. Next, for every y_i the simulator chooses a uniformly random $r_i \leftarrow \{0,1\}^{2\lambda}$, sets $w'_i = (k, r_i, 0^n, \psi = 0, y_i, i)$ and $\tilde{w_i}' \leftarrow \mathsf{GC.inp}(w'_i, s)$. The output of the simulator $\mathsf{GR.sim}$ consists of

$$(\ \tilde{D}',\ \ \tilde{C},\ \ \tilde{w}'_1, \ldots, \tilde{w}'_q\ \ ).$$

We next prove indistinguishability between the real and simulated outputs. Notice that

$$\mathsf{Real}[\mathsf{in}, \lambda] = (\ \tilde{D},\ \ \tilde{C},\ \ \tilde{w}_1, \ldots, \tilde{w}_q\ \ )$$

where $(\tilde{D}, k) \leftarrow \mathsf{GR.data}(1^\lambda, D)$, $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C)$, and we set $w_i = (k, r_i, x_i, \psi = 1, 0^m, i)$ for random $r_i \leftarrow \{0,1\}^{2\lambda}$ and $\tilde{w}_i \leftarrow \mathsf{GC.inp}(w'_i, s)$.

Claim 4.6. Denote $C = C[P, N, n, m, t, \lambda]$. If the scheme $\mathsf{GR1} = (\mathsf{GR1.prog}, \mathsf{GR1.inp}, \mathsf{GR1.eval})$ satisfies one-time security with persistent memory (Definition 4.2) then

$$(\ \tilde{D},\ \ C(w_1), \ldots, C(w_q)\ \ ) \overset{\text{comp}}{\approx} (\ \tilde{D}',\ \ C(w'_1), \ldots, C(w'_q)\ \ ).$$

Proof. Note that $C(w_i) = (\tilde{P}_i, \tilde{x}_i)$ and $C(w'_i) = (\tilde{P}_i, \tilde{x}'_i)$ where each $\tilde{P}_i$ is a freshly garbled version of the program P^+, $\tilde{x}_i$ is a garbled version of the input $(x_i, \psi = 1, 0^m)$, and $\tilde{x}'_i$ is a garble version of the input $(0^n, \psi = 0, y_i)$. The one-time simulation-security of the underlying GR1 scheme implies that:

$$(\ \tilde{D},\ \langle \tilde{P}_i,\ \tilde{x}_i \rangle_{i \in [q]}\ \ ) \overset{\text{comp}}{\approx} \mathsf{GR1.Sim}(1^\lambda, \mathsf{out}) \overset{\text{comp}}{\approx} (\ \tilde{D}',\ \langle \tilde{P}_i,\ \tilde{x}'_i \rangle_{i \in [q]}\ ).$$

This proves the claim. □

We now conclude that the simulated and real distributions are the indistinguishable:

$$(\ \tilde{D}',\ \ \tilde{C},\ \ \langle \tilde{w}'_i \rangle_{i \in [q]}\ \ ) \overset{\text{comp}}{\approx} (\ \tilde{D},\ \ \tilde{C},\ \ \langle \tilde{w}_i \rangle_{i \in [q]}\ \ )$$

This follows by the above claim in conjunction with the definition of correlated distributional indistinguishability on the circuit garbling scheme. We think of $\tilde{D} = \mathsf{aux}$ and $\tilde{D}' = \mathsf{aux}'$ as auxiliary input. □

4.3 Achieving Correlated Distributional Indistinguishability

Below we present two candidate constructions of reusable garbled circuits from obfuscation. The first one is very simple, but essentially relies on a "special-purpose" virtual black-box (VBB) obfuscation conjecture of some relevant functionality. The second construction is slightly more complex, and we can prove its security under an notion of obfuscation that we call *strong differing-inputs obfuscation* (and which generalizes differing-inputs obfuscation [BGI+12, BCP13, ABG+13]). Technically, the two assumptions needed to prove security of our two constructions are incomparable, and therefore we present both options.

Construction 1 (Obfuscating a "decrypt-then-evaluate" circuit). Let $\mathcal{PKE} = (\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{Decrypt})$ be a public-key encryption scheme. For any circuit $C : \{0,1\}^n \to \{0,1\}^m$ and decryption secret key sk we define the circuit $C^*[\mathsf{sk}](c)$ which computes $x = \mathsf{Decrypt}(\mathsf{sk}, c)$ and outputs $C(x)$. We define the circuit garbling scheme $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ as follows:

- $\mathsf{GC.circ}(1^\lambda, C)$: Generate $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$. Construct the circuit $C^* := C^*[\mathsf{sk}]$ from C as shown. Output $\tilde{C} \leftarrow \mathcal{O}(C^*)$ and $s := \mathsf{pk}$.
- $\mathsf{GC.inp}(x, s)$: Output $\tilde{x} \leftarrow \mathsf{Encrypt}(\mathsf{pk}, x)$.
- $\mathsf{GC.eval}(\tilde{C}, \tilde{x})$: Interpret $\tilde{C}$ as an obfuscated circuit and output $\tilde{C}(\tilde{x})$.

We simply conjecture that this construction is secure when instantiated with a standard-construction PKE and a "good" obfuscator, such as the candidate construction of [GGH+13].

Conjecture 4.7. There exists a CCA-secure public-key encryption scheme $\mathcal{PKE}$ and an obfuscator $\mathcal{O}$, for which the above construction GC is a *reusable garbled circuit with correlated distributional indistinguishability.*

As a "sanity check", it's easy to show that the conjecture holds if the attacker were only given black-box access to the circuit $C^*[\mathsf{sk}]$ rather than the obfuscated circuit $\tilde{C} \leftarrow \mathcal{O}(C^*[\mathsf{sk}])$. In particular, this follows from a sequence of hybrids where: (I) one-by-one, we modify the ith garbled input $\tilde{w}_i$ to be an encryption of 0 instead of w_i but make $C^*[\mathsf{sk}]$ return w_i when queried with $\tilde{w}_i$ (CCA-security), (II) we modify all the values returned by $C^*[\mathsf{sk}]$ from w_i to w'_i and switch aux to aux' (correlated ind. assumption) and (III) we go back one-by-one and modify ith garbled input to be an encryption of w'_i and $C^*[\mathsf{sk}]$ to decrypt correctly (CCA-security). We note that our conjecture does not formally require full VBB security and, for example, is not defined using a simulator. None of the known negative results for obfuscation seems to apply to the above conjecture, when instantiated with a "natural" public-key encryption (e.g., Cramer-Shoup construction [CS98]).

Construction 2 (Using Strong Differing-Inputs Obfuscators). Next we present a construction for reusable garbled circuits with correlated distributional indistinguishability, which we prove secure under a new stronger form of the differing-inputs obfuscation assumption. Differing-inputs obfuscation (diO) [BGI+12, BCP13, ABG+13] guarantees that for any two circuits C_0 and C_1, if it is difficult to find an input x on which $C_0(x) \neq C_1(x)$, then it should be hard to distinguish the obfuscation of C_0 from the obfuscation of C_1. We define a stronger version of the assumption which maintains the indistinguishability of the obfuscations even when given several points $x_1, \ldots, x_n$ on which the two circuits have different but *computationally indistinguishable outputs*, as long as it is hard to find any other inputs on which the circuits differ. In particular, feeding these inputs x_i to the obfuscated circuit does not help distinguish C_0 from C_1. We formalize this intuition in the following definition:

Definition 4.8. *A family of circuits $\mathcal{C}$ with a sampler $(C_0, C_1, x_1, \ldots, x_n, \mathsf{aux}_0, \mathsf{aux}_1) \leftarrow \mathsf{Sam}(1^\lambda)$, which samples $C_0, C_1 \in \mathcal{C}$ together with inputs $x_1, \ldots, x_n$ is said to be a* strong differing inputs family *if*

$$\{C_0(x_1), \ldots, C_0(x_n), \mathsf{aux}_0\} \overset{\text{comp}}{\approx} \{C_1(x_1), \ldots, C_1(x_n), \mathsf{aux}_1\}$$

and for all PPT adversaries $\mathcal{A}$ there is a negligible function ε such that

$$\Pr\left[\begin{array}{c|c} C_0(x) \neq C_1(x) & (C_0, C_1, x_1, \ldots, x_n, \mathsf{aux}_0, \mathsf{aux}_1) \leftarrow \mathsf{Sam}(1^\lambda) \\ \wedge \;\; x \notin \{x_1, \ldots, x_n\} & x \leftarrow \mathcal{A}(1^\lambda, C_0, C_1, x_1, \ldots, x_n, \mathsf{aux}_0, \mathsf{aux}_1) \end{array}\right] \leq \varepsilon(\lambda).$$

Definition 4.9 (Strong Differing-Inputs Obfuscator (sdiO))**.** *A PPT algorithm $\mathcal{O}$ satisfying correctness is a* strong differing-inputs obfuscator (sdiO) *for a strong differing-inputs family $\mathcal{C}$ with a sampler Sam, if for all PPT distinguisher algorithms $\mathcal{D}$, there is a negligible function ε such that*

$$|\Pr[\mathcal{D}(1^\lambda, \mathcal{O}(1^\lambda, C_0), x_1, \ldots, x_n, \mathsf{aux}_0) = 1] - \Pr[\mathcal{D}(1^\lambda, \mathcal{O}(1^\lambda, C_1), x_1, \ldots, x_n, \mathsf{aux}_1) = 1]| \leq \varepsilon(\lambda),$$

where $(C_0, C_1, x_1, \ldots, x_n, \mathsf{aux}_0, \mathsf{aux}_1) \leftarrow \mathsf{Sam}(1^\lambda)$.

The recent work of [GGHW13] gives some evidence that general-purpose diO relative to arbitrary auxiliary input is unlikely to exist. The same negative result would hold for the stronger notion of general-purpose sdiO. However, the counterexample relies on highly contrived auxiliary input which is itself an obfuscation. Therefore, it still makes sense to assume diO and sdiO holds when the auxiliary input $\mathsf{aux}_0, \mathsf{aux}_1$ and the samples x_i have some concrete structure. In our case, the auxiliary input and the points $x_1, \ldots, x_n$ will simply contain a public-key of an encryption scheme and some simulated NIZK proofs. Therefore, the results of [GGHW13] do not apply to sdiO for the restricted type of strong differing input circuit families ($\mathcal{C}$,Sam) that we rely on.

The construction of reusable garbled circuits from sdiO is similar to (but simpler than) the one from Section 3.3. Let $\mathcal{O}$ be a strong differing-inputs obfuscator, $\mathcal{PKE} = (\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{Decrypt})$ be a semantically secure encryption scheme, and $\Pi = (K, P, V)$ be a simulation-sound NIZK scheme. For any circuit $C : \{0,1\}^n \to \{0,1\}^m$ we define a circuit $C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b, \mathsf{sk}](c_0, c_1, \pi)$, which takes as input two ciphertexts c_0, c_1 and an NIZK proof π for $(\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \in L_{EQ}$, where L_{EQ} is defined as in the previous section, and computes the following:

$\underline{C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, b, \mathsf{sk}](c_1, c_2, \pi)}$:
1. Verify that π is a proof of $(\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \in L_{EQ}$ by running $V(\sigma, (\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2), \pi)$. If this rejects, output $\perp$.
2. Compute $x = \mathsf{Decrypt}(\mathsf{sk}, c_b)$. Output $C(x)$.

We define the circuit garbling scheme $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ as follows:

- $\mathsf{GC.circ}(1^\lambda, C)$: Generate $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\lambda)$, $\sigma \leftarrow K(1^\lambda)$. Construct the circuit $C^* := C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, 0, \mathsf{sk}_0]$ from C as shown above. Output $\tilde{C} \leftarrow \mathcal{O}(1^\lambda, C^*)$ and $s := (\sigma, \mathsf{pk}_0, \mathsf{pk}_1)$.
- $\mathsf{GC.inp}(x, s)$: Compute $c_0 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_0, x; r_0)$, $c_1 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_1, x; r_1)$ and $\pi \leftarrow P(\sigma, (\mathsf{pk}_0, \mathsf{pk}_1, c_0, c_1), (r_0, r_1))$ is a NIZK that $(\mathsf{pk}_0, \mathsf{pk}_1, c_0, c_1) \in L_{EQ}$. Output $\tilde{x} \leftarrow (c_0, c_1, \pi)$.
- $\mathsf{GC.eval}(\tilde{C}, \tilde{x})$: Interpret $\tilde{C}$ as an obfuscated circuit and output $\tilde{C}(\tilde{x})$.

We note that in the above construction we need only simulation sound NIZK scheme, which does not have to provide statistical security. In particular, previously we needed statistical security to make sure that there are no inputs on which two different circuits (decrypting with sk_0 vs sk_1) differ. Now, it suffices that such inputs are hard to find.

We define a class of *relevant* circuit families $(\mathcal{C}, \mathsf{Sam})$ where:

- $\mathcal{C}$ consists of circuits of the form $C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, b, \mathsf{sk}]$ as described above.
- $(C_0, C_1, \tilde{x}_1, \ldots, \tilde{x}_n, \mathsf{aux}_0, \mathsf{aux}_1) \leftarrow \mathsf{Sam}(1^\lambda)$ consists of setting $C_0 = C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b = 0, \mathsf{sk}_0]$, $C_1 = C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b = 1, \mathsf{sk}_1]$ where σ is a simulated CRS chosen via $(\sigma, \tau) \leftarrow S_1(1^\lambda)$ with trapdoor τ.
- The inputs $\tilde{x}_i = (c_0^i, c_1^i, \pi^i)$ consist of encryptions $c_b^i \leftarrow \mathsf{Encrypt}(\mathsf{pk}_b, x_b^i)$ and simulated proofs $\pi^i \leftarrow S_2(\sigma, \tau, c_0^i, c_1^i)$.
- The encrypted messages x_b^i and the auxiliary information $\mathsf{aux}_0, \mathsf{aux}_1$ are chosen independently of σ, τ.

Although this may seem like a complicated class of relevant circuit families, we notice that when instantiated with standard construction cryptosystem and and NIZK, the above restricted class already prevent counterexamples such as the one of [GGHW13] from going through. In particular, such counterexamples crucially rely on the auxiliary input (or the additional values x_i) containing and obfuscated circuit which has embedded secret information that can be used to come up with an input on which C_0, C_1 would distinguishably differ. In our case, the only way to come up with such an input is to know the NIZK trapdoor τ but we only give out some selected few simulated proofs π^i using τ and do not provide any other information about it in any other context. It seems reasonable to postulate the existence of sdiO for all relevant differing-inputs families relative to some standard-construction encryption scheme $\mathcal{PKE}$ and simulation sound NIZK scheme Π.

Theorem 4.10. *If Π is simulation sound NIZK, and $\mathcal{PKE}$ is a semantically secure public-key encryption scheme and $\mathcal{O}$ is a strong differing-inputs obfuscator for relevant differing-inputs circuit families, then the above construction GC is a* reusable garbled circuit with correlated distributional indistinguishability.

Proof. Let C be a circuit (ensemble) and D and D' be efficiently samplable input distributions (ensembles) such that it hold that

$$\langle C(x_1^0), \ldots, C(x_n^0), \mathsf{aux}_0 \rangle \overset{\text{comp}}{\approx} \langle C(x_1^1), \ldots, C(x_n^1), \mathsf{aux}_1 \rangle.$$

where $\langle x_1^0, \ldots, x_n^0, \mathsf{aux}_0 \rangle \leftarrow D(1^\lambda), \langle x_1^1, \ldots, x_n^1, \mathsf{aux}_1 \rangle \leftarrow D'(1^\lambda)$. We will show that

$$\langle \tilde{C}, \tilde{x}_1^0, \ldots \tilde{x}_n^0, \mathsf{aux}_0 \rangle \overset{\text{comp}}{\approx} \langle \tilde{C}, \tilde{x}_1^1, \ldots \tilde{x}_n^1, \mathsf{aux}_1 \rangle$$

where $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C),\ \tilde{x}_i^0 \leftarrow \mathsf{GC.inp}(x_i^0, s)\ ,\ \tilde{x}_i^1 \leftarrow \mathsf{GC.inp}(x_i^1, s)$.

We consider the following sequence of hybrid distributions to show the indistinguishability of the above distributions:

- Hyb_0: This is the distribution $\langle \tilde{C}, \tilde{x}_1^0, \ldots \tilde{x}_n^0, \mathsf{aux}_0 \rangle$, where $\tilde{C} \leftarrow \mathcal{O}(C^*)$ for $C^* := C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, 0, \mathsf{sk}_0]$, and $\tilde{x}_i^0 = (c_0^i, c_1^i, \pi^i)$ where $c_b^i \leftarrow \mathsf{Enc}(\mathsf{pk}_b, x_i^0; r_b^i)$, $\pi^i \leftarrow P(\sigma, (\mathsf{pk}_0, \mathsf{pk}_1, c_0^i, c_1^i), (r_0^i, r_1^i))$ for all $1 \le i \le n$.
- Hyb_1: In this hybrid, we compute the proofs π^i using the NIZK simulators (S_1, S_2), i.e., $(\sigma, \tau) \leftarrow S_1(1^\lambda)$, $\pi^i \leftarrow S_2(\sigma, \tau, c_0^i, c_1^i)$ for $1 \le i \le n$.

  The indistinguishability $\mathsf{Hyb}_0 \overset{\text{comp}}{\approx} \mathsf{Hyb}_1$ follows from the computational zero-knowledge of property of the NIZK Π.

- Hyb_2: In this hybrid, we also change the way we compute the garbled inputs by setting $c_1^i \leftarrow \mathsf{Encrypt}(\mathsf{pk}_1, x_i')$ to be an encryption of x_i^1 rather than x_i^0, for all $1 \le i \le n$,. The NIZK CRS and proofs π_i are still computed using the NIZK simulator.

  The indistinguishability $\mathsf{Hyb}_1 \overset{\text{comp}}{\approx} \mathsf{Hyb}_2$ follows from the semantic security of the encryption scheme $\mathcal{PKE}$ since sk_1 does not appear anywhere in the garbled circuit.

- Hyb_3: In this hybrid, we change the way we compute the garbled circuit by obfuscating $C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b = 1, \mathsf{sk}_1]$ with the bit $b = 1$ and the secret key sk_1 (instead of $b = 0$ and sk_0). That is, we set $\tilde{C}' \leftarrow \mathcal{O}(C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b = 1, \mathsf{sk}_1])$. We also change the auxiliary input from aux_0 to aux_1 We claim that:

  $$\mathsf{Hyb}_2 \overset{\text{comp}}{\approx} \mathsf{Hyb}_3.$$

  This follows from the strong differing-inputs obfuscation property. Firstly, we claim that $(\mathcal{C}, \mathsf{Sam})$ is a relevant differing-input family where $\mathcal{C} = \{C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b, \mathsf{sk}]\}$ and

  $$(C_0 = C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b = 0, \mathsf{sk}_0], C_1 = C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b = 1, \mathsf{sk}_1], \tilde{x}_1, \ldots, \tilde{x}_n, \mathsf{aux}_0, \mathsf{aux}_1) \leftarrow \mathsf{Sam}(1^\lambda)$$

  is defined by sampling $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\sigma, \tau) \leftarrow S_1(1^\lambda)$, $\langle x_1^0, \ldots, x_n^0, \mathsf{aux}_0\rangle \leftarrow D(1^\lambda), \langle x_1^1, \ldots, x_n^1, \mathsf{aux}_1\rangle \leftarrow D'(1^\lambda)$ and setting $\tilde{x}_i = (c_0^i, c_1^i, \pi^i)$ where $c_b^i \leftarrow \mathsf{Enc}(\mathsf{pk}_b, x_i^b)$ and $\pi^i \leftarrow S_2(\sigma, \tau, c_0^i, c_1^i)$.

  This follows since

  $$(\langle C_0(\tilde{x}_i) = x_i^0\rangle_{i\in[q]}, \mathsf{aux}_0) \overset{\text{comp}}{\approx} (\langle C_1(\tilde{x}_i) = x_i^1\rangle_{i\in[q]}, \mathsf{aux}_1)$$

  by assumption. Furthermore, coming up with an input $\tilde{x} \notin \{\tilde{x}_1, \ldots, \tilde{x}_n\}$ such that $C_0(\tilde{x}) \neq C_1(\tilde{x})$ requires coming up with a valid NIZK proof for a new false statement, which is hard by the simulation-soundness security of the NIZK even given $C_0, C_1, \langle \tilde{x}_i\rangle, \mathsf{aux}_0, \mathsf{aux}_1$.

  Now that we showed $(\mathcal{C}, \mathsf{Sam})$ is a relevant differing-input family, we can rely on sdiO security of the obfuscator $\mathcal{O}$ to argue that $\mathsf{Hyb}_2 \overset{\text{comp}}{\approx} \mathsf{Hyb}_3$. In particular one cannot distinguish $(\mathcal{O}(1^\lambda, C_0), \langle \tilde{x}_i'\rangle_{i\in[q]}, \mathsf{aux}_0)$ from $(\mathcal{O}(1^\lambda, C_1), \langle \tilde{x}_i'\rangle_{i\in[q]}, \mathsf{aux}_1)$ where $\tilde{x}_i'$ are chosen as in hybrid 2.

- Hyb_4 to Hyb_6: These hybrid are symmetric to the changes introduced by Hyb_1 to Hyb_3 in a slightly different order. In Hyb_4 we change the encryptions to $c_0^i = \mathsf{Encrypt}(\mathsf{pk}_0, x_i^1)$ (semantic security), in Hyb_5 we switch to honestly generated NIZK proofs π^i (ZK property) and finally in Hyb_6 we switch back to an obfuscation of the circuit $C_0 = C^*[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b = 0, \mathsf{sk}_0]$ (sdiO). The last Hyb_6 is just the distribution $\langle \tilde{C}, \tilde{x}_1^1, \ldots \tilde{x}_n^1\rangle$. By the same arguments as described above, we get

  $$\mathsf{Hyb}_3 \overset{\text{comp}}{\approx} \mathsf{Hyb}_6.$$

Combining the above, we get that $\mathsf{Hyb}_0 \overset{\text{comp}}{\approx} \mathsf{Hyb}_6$, which completes the proof of the theorem. □

Summary. Combining the above with Theorem 4.3 and 4.5, and that statistically simulation sound NIZK can be constructed from statistically sounds NIZK [GGH+13], we get the following corollary.

Corollary 4.11. *Assuming the existence of sdiO for all relevant differing-inputs families, selectively secure IBE and statistically sound NIZK, there exists a reusable garbled-RAM scheme with persistent memory (Definition 4.2).*

5 Compact Reusable GRAM

The constructions of reusable GRAM so far correspond to the "*basic*" (Section 3) and "*middle*" (Section 4) solution from the introduction. In particular, both solutions require a "one-time pre-processing" where the client has to work as hard as evaluating the program in order to garble the program. This garbling only needs to be performed once per program and can then be used to evaluate the garbled program on many garbled inputs, providing amortized efficiency. We now give candidate constructions for reusable GRAM with persistent memory, which is also *compact*. That is, the time that it takes to garble a program (and the size of the resulting garbled RAM program) do not depend on the running time of that program, but only on its description length. This corresponds to the "best case" solution from the introduction.

Let us begin by considering the reason why the previous constructions above are not compact. In these constructions, we garble a program P for many uses by writing down (and then garbling) a description of circuit $C[P]$ that outputs the one-time program-garbling of (roughly) the same program P. But this circuit is as large as the run-time of the one-time garbling procedure, which is as large as the run-time of P itself. To get around this problem, we would like to represent $C[P]$ by a much smaller circuit that "can compute the same function". Here we rely on the fact that the one-time GRAM schemes from the literature are *bit-wise compact*: there is a small circuit $C_{bit}[P]$ that takes an additional input index j and outputs the j'th bit in the description of the one-time garbled $\tilde{P}$. We note that for the one-time GRAM schemes from the literature are bit-wise compact, see Appendix B for details.

To achieve compactness, our approach is therefore to garble this small circuit $C_{bit}[P]$ instead of the large $C[P]$. This does not quite work, however, since this small circuit needs a huge input in order to output all of $\tilde{P}$ (i.e., all the indexes $1, 2, \ldots, |\tilde{P}|$). If we were to just garble $C_{bit}[P]$, we would be losing on input-garbling all the gains from program-garbling. Instead, we need a special circuit-garbling routine for the original $C[P]$, whose complexity depends only on the size of the smaller $C_{bit}[P]$. We call this notion of efficiency *compactness* for circuit-garbling, and note that this is similar to, but not the same as, the compactness that we seek for garbled RAM.

5.1 Definitions

Below we define our goal (compact reusable GRAM) and our tools (bitwise-compact one-time GRAM and compact garbled circuits).

Compact circuit-garbling. The essence of this notion is that the circuit-garbling procedure gets as input the small circuit C_{bit} rather than C itself, and works in time $|C_{bit}| \cdot \mathrm{poly}(\lambda)$. However, importantly the $\mathsf{GC.inp}$ and $\mathsf{GC.eval}$ routines still work relative to the original circuit C and not the small C_{bit}. To make this formal, we first establish notations to relate the circuits C and C_{bit}.

Definition 5.1 (Bitwise representation)**.** A function $f_{bit} : \{0,1\}^n \times [m] \to \{0,1\}$ is the *bitwise representation* of another function $f : \{0,1\}^n \to \{0,1\}^m$ if for any $x \in \{0,1\}^n$ and $j \in [m]$ it holds that $f_{bit}(x, j) = f(x)_j$.

Similarly a circuit C_{bit} is a bitwise representation of another circuit C if the function that C_{bit} computes is the bitwise representation of the function computed by C.

Definition 5.2 (Compact Garbled Circuits)**.** A Compact garbled circuit scheme consists of three procedures $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ as follows:

- $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C_{bit})$: On input circuit C_{bit}, output a garbled circuit $\tilde{C}$ and key s.
- $\tilde{x} \leftarrow \mathsf{GC.inp}(x, s)$: Gets an input x and the same key s. Outputs the garbled input $\tilde{x}$.

- $y \leftarrow \mathsf{GC.eval}(\tilde{C}, \tilde{x})$: Gets a garbled circuit $\tilde{C}$ and matching input $\tilde{x}$, and computes the output.

We require that for any two circuits C, C_{bit} with C having m-bit outputs and C_{bit} a bitwise representation of C, and any input $x \in \{0,1\}^n$, setting $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C_{bit})$ and $\tilde{x} \leftarrow \mathsf{GC.inp}(x, s)$ we get $\mathsf{GC.eval}(\tilde{C}, \tilde{x}) = C(x)$. We also require that $\mathsf{GC.circ}$ works in time $|C_{bit}| \cdot \mathrm{poly}(\lambda)$, $\mathsf{GC.inp}$ works in time $n \cdot \mathrm{poly}(\lambda)$, and $\mathsf{GC.eval}$ works in time $\widetilde{O}(m \cdot |C_{bit}|) \cdot \mathsf{poly}(\lambda)$.

Definition 5.3 (Compact GRAM)**.** A garbled RAM scheme GR as in Definition 3.1 or Definition 4.1 is *compact* if the procedure $\tilde{P} \leftarrow \mathsf{GR.prog}(P, \ldots)$ runs in time $\widetilde{O}(|P| + n + m) \cdot \mathrm{poly}(\lambda, \log N, \log t)$, i.e., effectively independent of the program's running time t.[10] This also serves as a bound on the size of the garbled program $\tilde{P}$.

Definition 5.4 (Bitwise Compact GRAM)**.** A garbled RAM scheme GR as in Definition 3.1 or Definition 4.1 is *bit-wise compact* if the bitwise representation of $\mathsf{GR.prog}(P, \ldots)$ can be computed by a circuit of size $\widetilde{O}(|P| + n + m) \cdot \mathrm{poly}(\lambda, \log N, \log t)$.

We stress that bit-wise compactness is a significantly weaker property than compactness, as the total size of the garbled program $\tilde{P}$ may still depend on t. Indeed, the one-time garbled RAM schemes from the literature already turn out to satisfy the compactness property. See Appendix B for an overview of why the following theorem holds.

Theorem 5.5 ([LO13a, GHL+14])**.** *Assuming the existence of selectively-secure identity-based encryption (IBE), there exist* bitwise compact *garbled RAM schemes (with or without persistent memory) with* one-time *security.*

5.2 Constructing Compact Reusable GRAM

We observe that a slight variant of the constructions from 3.6 and 4.5 already provide compactness if the underlying building blocks are efficient enough. Note that if the underlying one-time GRAM scheme is bitwise compact, then the circuit $\mathbf{C}[\mathbf{P}, \mathbf{n}, \mathbf{m}, \mathbf{t}, \lambda]$ described in Section 3.2 has a bitwise representation $\mathbf{C_{bit}}[\mathbf{P}, \mathbf{n}, \mathbf{m}, \mathbf{t}, \lambda]$ of size $(|P| + n + m) \cdot \mathrm{poly}(\lambda, \log N, \log t)$. The same observation applies to Section 4.2 and the circuit $\mathbf{C}[\mathbf{P}, \mathbf{N}, \mathbf{n}, \mathbf{m}, \mathbf{t}, \lambda]$ there.

Observation 5.6. In the constructions from Section 3.2 and Section 4.2, if the underlying one-time GRAM scheme is bitwise compact and the underlying reusable circuit garbling scheme is compact (and if the program-garbling routine is modified so as to produce and garble the bitwise representation $C_{bit}[P, \ldots]$ of size-$(|P| + n + m) \cdot \mathrm{poly}(\lambda, \log N, \log t)$ rather than $C[P, \ldots]$ itself), then the resulting garbled RAM scheme is compact.

Since the one-time garbled-RAM constructions from the literature [LO13a, GHRW14] are bitwise compact (cf. Appendix B), we only need to describe how to realize compact reusable garbled circuits.

5.3 Achieving Compact Reusable Garbled Circuits

We can take advantage of our concrete Construction 1 & 2 in Section 4.3 of reusable garbled circuits with correlated-distributional indistinguishability based on obfuscation. In particular, we claim that these constructions can be made compact without a significant change to the construction or assumption. In contrast, we do not know how to make the construction of reusable garbled circuits with basic distributional indistinguishability from iO compact without moving to the above stronger assumptions. We start with Construction 1 in Section 4.3, and define the optimized variant as follows.

[10] In the case of Definition 3.1 we do not have the parameter N, so we just use the convention $N = 0$.

Construction 1'. Given a circuit $C : \{0,1\}^n \to \{0,1\}^m$ with a short bitwise representation $C_{bit} : \{0,1\}^n \times [m] \to \{0,1\}$ and a decryption key sk we define the circuit $C^*_{bit}[\mathsf{sk}](c,i)$ which computes $x = \mathsf{Decrypt}(\mathsf{sk}, c)$ and outputs $C_{bit}(x,i)$. Notice that the size of $|C^*_{bit}[\mathsf{sk}]|$ is only proportional to $|C_{bit}|$. We define the the optimized circuit garbling scheme $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$:

- $\mathsf{GC.circ}(1^\lambda, C_{bit})$: Generate $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$. Construct the circuit $C^* := C^*_{bit}[\mathsf{sk}]$ as above. Output $\tilde{C} \leftarrow \mathcal{O}(C^*)$ and $s := \mathsf{pk}$.
- $\mathsf{GC.inp}(x, s)$: Output $\tilde{x} \leftarrow \mathsf{Encrypt}(\mathsf{pk}, x)$.
- $\mathsf{GC.eval}(\tilde{C}, \tilde{x})$: Interpret $\tilde{C}$ as an obfuscated circuit. Output $y = (\tilde{C}(\tilde{x}, 1), \ldots, \tilde{C}(\tilde{x}, m))$.

Notice that the running time of $\tilde{C} \leftarrow \mathsf{GC.circ}(1^\lambda, C)$ and the size of $\tilde{C}$ is at most $|C_{bit}|\mathsf{poly}(\lambda)$. We define a new conjecture which seems qualitatively similar to Conjecture 4.7.

Conjecture 5.7. There exists a CCA-secure public-key encryption scheme $\mathcal{PKE}$ and an obfuscator $\mathcal{O}$, for which the above construction GC is a *copact reusable garbled circuit with correlated distributional indistinguishability.*

We also provide a construction for compact reusable garbled circuits, the security of which can be based on strong differing-inputs obfuscation.

Construction 2'. Let $\mathcal{O}$ be a strong differing-inputs obfuscator, $\mathcal{PKE} = (\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{Decrypt})$ be a semantically secure encryption scheme, and $\Pi = (K, P, V)$ be a simulation sound NIZK scheme. Given a circuit $C : \{0,1\}^n \to \{0,1\}^m$, let $C_{bit} : \{0,1\}^n \times [m] \to \{0,1\}$ be the compact representation for C as defined above. We define a circuit $C^*_{bit}[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b, \mathsf{sk}](c_0, c_1, \pi, i)$, which takes as input two ciphertexts c_0, c_1, a NIZK proof π for $(\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \in L_{EQ}$, where L_{EQ} is defined as in the previous section, and a plaintext i, and computes the following:

$C^*_{bit}[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, b, \mathsf{sk}](c_1, c_2, \pi)$:
1. Verify that π is a proof of $(\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \in L_{EQ}$ by running $V(\sigma, (\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2), \pi)$. If this rejects, output $\perp$.
2. Compute $x = \mathsf{Decrypt}(\mathsf{sk}, c_b)$. Output $C_{bit}(x, i)$.

We define the circuit garbling scheme $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ as follows:

- $\mathsf{GC.circ}(1^\lambda, C)$: Generate $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\lambda)$, $\sigma \leftarrow K(1^\lambda)$. Construct the circuit $C^* := C^*_{bit}[\sigma, \mathsf{pk}_0, \mathsf{pk}_1, b, \mathsf{sk}_0]$ from C as shown above. Output $\tilde{C} \leftarrow \mathcal{O}(C^*)$ and $s := (\sigma, \mathsf{pk}_0, \mathsf{pk}_1)$.
- $\mathsf{GC.inp}(x, s)$: Compute $c_0 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_0, x; r_0)$, $c_1 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_1, x; r_1)$ and $\pi \leftarrow P(\sigma, (\mathsf{pk}_0, \mathsf{pk}_1, c_0, c_1), (r_0, r_1))$ is a NIZK that $(\mathsf{pk}_0, \mathsf{pk}_1, , c_0, c_1) \in L_{EQ}$. Output $\tilde{x} \leftarrow (c_0, c_1, \pi)$.
- $\mathsf{GC.eval}(\tilde{C}, \tilde{x})$: Interpret $\tilde{C}$ as an obfuscated circuit and output $y = (\tilde{C}(\tilde{x}, 1), \ldots, \tilde{C}(\tilde{x}, m))$.

Theorem 5.8. *If Π is simulation sound NIZK, and $\mathcal{PKE}$ is a semantically secure encryption scheme and $\mathcal{O}$ is a strong differing-inputs obfuscator for relevant circuit families, then the above construction GC is a* compact reusable garbled circuit with correlated distributional indistinguishability.

The proof mirrors that of Theorem 4.10.

Summary. To summarize, we get the following corollary.

Corollary 5.9. *If there exist strong differing-inputs obfuscators for relevant circuit families, selectively secure IBE schemes, and statistically sound NIZKs then there exists a reusable garbled-RAM scheme (with or without persistent memory) which is also compact.*

6 Extensions and Applications

Program Privacy. Our default definitions of garbled RAM do not include program privacy, and the garbled program $\tilde{P}$ may reveal information about the code of the actual program P to the server. There are several simple standard techniques that can be employed to add program privacy to our constructions. Firstly, we can garble a program P_{uni} which is the *universal RAM* that runs for some fixed number of steps t. The code of the actual program P that we want to execute can then be provided as part of the input to P_{uni} or, in the case of persistent memory, it can be included as part of the initial memory data D. Alternatively, to avoid sending the code of the program P with each input in the case without persistent memory, we can use the following approach of [GKP+13b]: Instead of garbling the program P, we encrypt it to get a cipehrtext $c_P = \mathsf{Enc}_k(P)$. We then garble a program $Q_{enc}[c_P]$ that has c_P hard wired in it, and on input (k, x) it decrypts c_P with key k, interpret the result as a program, and run that program on input x. Notice that the input size for $Q_{enc}[c_P]$ is independent of the size of P itself, and that the description of $Q_{enc}[c_P]$ does not reveal anything about the actual program P.

Output Privacy. Our default definition of garbled RAM assumes that the server who evaluates the garbled program on the garbled input learns the output y of the program. This might be useful in some scenarios, but in other cases the output y is intended for the client and the server simply sends it back to the client. In such cases, we may also want the output y of the program to remain secret from the server and only be revealed to the client. We can achieve *output privacy* by simply garbling an augmented program P_{outEnc} which gets as input (k, x), evaluates $y = P(x)$ and outputs an encryption $c = \mathsf{Enc}_k(y)$, where $(\mathsf{Enc}, \mathsf{Dec})$ is some (one-time) encryption scheme (e.g., one-time pad). The client garbles inputs (x_i, k_i) where k_i is a fresh key for encryption, the server evaluates the garbled program on the garbled input to get back $c_i = \mathsf{Enc}_{k_i}(y_i)$ where $y_i = P(x_i)$, sends c_i to the client, and the client decrypts $y_i = \mathsf{Dec}_{k_i}(c_i)$. Notice that the entire view of the server can be simulated given the values c_i and therefore the server learns nothing about the inputs x_i or the outputs y_i of the program executions.

Verifiable Computation. In the above scenario, where the program outputs y_i are intended for the client, we may also want to add *verifiability*, where the client is sure that the received outputs y_i are indeed the correct outputs of the computation. To do so, we can garble an augmented program P_{outAuth} which gets as input (k, x), evaluates $y = P(x)$ and outputs (y, σ) where σ is an authentication-tag $\sigma = \mathsf{MAC}_k(y)$ for some (one-time) message-authentication code MAC. The client garbles inputs (x_i, k_i) where k_i is a fresh key for the MAC, the server evaluates the garbled program on the garbled input to get back (y_i, σ_i) which it sends to the client, and the client verifies $\sigma_i = \mathsf{MAC}_{k_i}(y_i)$. Notice that the entire view of the server can be simulated given the values (y_i, σ_i) and therefore the server cannot come up with a valid tag σ'_i for any $y'_i \neq y_i$. This gives us verifiable computation. Furthermore, we can always combine privacy and authentication by garbling an augmented program $P_{\mathsf{outEncAuth}}$ which encrypts the output and then authenticates the ciphertext.

Input-Specific Run-Time. Our default notion of garbled RAM assumes that the program P has some fixed worst-case run-time $t = t(n)$ for all inputs x of size n, and and the running time

of the server during each evaluation is proportional to t. However, a program might have vastly different run-times for different inputs, and we would like the server's work when evaluating the garbled program $\tilde{P}$ on garbled inputs $\tilde{x}$ to only depend on the *input-specific run-time* of $P(x)$ rather than the worst-case run-time t. Of course, this means that we have to leak the input-specific run-time of $P(x)$, but the goal is to *only* leak this information and nothing else. The works of [GKP+13b, GKP+13b] show how to do this in the case of Turing Machine computation, where the server's run-time is proportional to the input-specific Turing-Machine run-time, and we can use some of the same techniques to get analogous results for RAM computation.

In the setting *without persistent memory*, the high-level idea goes as follows. To garble a program P, we separately garble $\ell = \log(t)$ programs $P_1, P_1, \ldots, P_\ell$ where each program $P_j(x)$ runs $P(x)$ for 2^j steps, and if the computation completes returns the output else indicates that the computation is incomplete. The server gets these independently garbled programs $\tilde{P}_1, \ldots, \tilde{P}_\ell$. To garble an input x, the client just creates ℓ independently garbled inputs $\tilde{x}_j$ for each of the reusable garbled RAM programs $\tilde{P}_j$ and the server evaluates them one by one (starting with $j = 1$) until the first one terminates. If the input-specific run-time is t_x then the time to evaluate all of the additional programs only introduces a constant amount of overhead and therefore the server's run-time is $\widetilde{O}(t_x)$. Notice that if we use a *compact* garbled RAM scheme, then the client never does work proportional to t but only proportional to $\log(t)$ where t is the worst-case run-time

In a setting *with persistent memory*, the solution becomes more complex for several reasons:

- The most serious issue is that our solution of reusable GRAM with persistent memory only allows us to garble a single program per database, and moreover we assume that this program always takes the same number of steps to run (since we are garbling a circuit whose output length depends on the run-time of the program). This issue can be addressed using ideas similar to our compact garbling scheme from Section 5. In particular, we can reusably garble a single circuit $C[P, \ldots]$ with a similar description as before but taking an additional input $t_{cur} \in \{1, \ldots, t\}$ and outputting a variable-length output consisting of $(\tilde{P}, \tilde{x})$ where $\tilde{P}$ runs for t_{cur} steps. In particular, this circuit is "bitwise-compact" even if the total number of output bits is not fixed. The client can sequentially garble her input with the values $t_{cur} = 2^j$ for $j = 1, \ldots, \log t$ until she hits a value t_{cur} for which the computation completes. Several minor issues remain.

- One issue is that each program execution which does not complete must "clean up after itself." Namely if execution does not halt within t_{cur} steps then it needs to restore the memory to its original state. This can be done without increasing the running time by too much.

- Yet another problem is that when garbling the i'th input, the client must know the total number of CPU steps executed so far (see generalized definition of one-time garbled RAM in [GHL+14]). Here we must revert to an interactive solution, where the client garbles these inputs one at a time and waits for the server to tell it whether the computation completed before it garbles the same input again (or a different input if the computation was completed).

The above outlines the high-level ideas and we defer a formal treatment of this to future work.

Applications to MPC. We have discussed garbled RAM in the context of outsourcing computation where the program specification, persistent memory/databse, and inputs are all chosen by a single client. However, we can also employ garbled RAM in the setting of two-party or multi-party computation (MPC) where these values belong to multiple parties. In particular, we can run standard MPC protocols to garble memory data D and program P where the underlying data belongs to several mutually distrustful parties. Later, the parties can run MPC protocol(s) to create

garbled inputs x_i, which may also depend on the inputs of several parties. The main advantage is that there can now be one designated (and untrusted) party that does the work of evaluating the garbled program on the garbled input in time proportional to the RAM computation, and all other parties only need to work in time proportional to the input size. This is in contrast to standard MPC approaches where all parties work as hard as the circuit size of the program evaluation, or the work of [GKK+12] where all parties work as hard as the RAM complexity of the computation.

7 Conclusions

We have shown how to privately outsource RAM computation from a weak client to a more powerful server via reusable garbled RAM schemes. Our main contribution was to reduce the problem of reusable garbled RAM into seemingly simpler problems dealing with reusable garbled circuits. In doing so, we introduced new notions of security for such garbled circuit that we call "distributional indistinguishability" and "correlated distributional indistinguishability" which may be of independent interest and seem to allow for greater (output-size independent) efficiency than the stronger simulation-based security. Lastly, we showed how to construct such schemes under obfuscation-based assumptions. The main open problem is to provide constructions of such reusable garbled circuits under weaker assumptions. Ideally, such constructions would avoid obfuscation altogether, but a more limited goal would be to get "correlated distributional indistinguishability" from indistinguishability obfuscation.

8 Acknowledgments

The authors would like to thank Yael Tauman Kalai, Nir Bitansky and Omer Paneth for enlightening initial discussions on the topics of this work.

References

[ABG+13] Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, and Mark Zhandry. Differing-inputs obfuscation and applications. Cryptology ePrint Archive, Report 2013/689, 2013. `http://eprint.iacr.org/`.

[AGVW13] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In Canetti and Garay [CG13], pages 500–518.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In Boneh et al. [BRF13], pages 111–120.

[BCP13] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. Cryptology ePrint Archive, Report 2013/650, 2013. `http://eprint.iacr.org/`.

[BFR+13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. *IACR Cryptology ePrint Archive*, 2013:356, 2013.

[BGI+12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[BRF13] Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors. *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*. ACM, 2013.

[BSCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In Canetti and Garay [CG13], pages 90–108.

[BSCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In Robert D. Kleinberg, editor, *ITCS*, pages 401–414. ACM, 2013.

[CG13] Ran Canetti and Juan A. Garay, editors. *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*. Springer, 2013.

[CIJ+13] Angelo De Caro, Vincenzo Iovino, Abhishek Jain, Adam O'Neill, Omer Paneth, and Giuseppe Persiano. On the achievability of simulation-based security for functional encryption. In *CRYPTO*, 2013.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[CR73] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.

[CS98] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1998.

[FLS99] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple non-interactive zero knowledge proofs under general assumptions. *SIAM Journal of Computing*, 29(1):1–28, 1999.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. volume 2013, page 451, 2013. To appear in FOCS 2013.

[GGHW13] Sanjam Garg, Craig Gentry, Shai Halevi, and Daniel Wichs. On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. Cryptology ePrint Archive, Report 2013/860, 2013. `http://eprint.iacr.org/`.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. EUROCRYPT, 2014.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Garbled ram revisited, part i. Cryptology ePrint Archive, Report 2014/082, 2014. `http://eprint.iacr.org/`.

[GKK+12] 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 *CCS*, 2012.

[GKP+13a] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In Canetti and Garay [CG13], pages 536–553.

[GKP+13b] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Boneh et al. [BRF13], pages 555–564.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[LO13a] Steve Lu and Rafail Ostrovsky. How to garble ram programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer, 2013.

[LO13b] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. Presentation in TCC 2013 rump session, 2013.

[LO14] Steve Lu and Rafail Ostrovsky. Garbled ram revisited, part ii. Cryptology ePrint Archive, Report 2014/083, 2014. `http://eprint.iacr.org/`.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage. In *STOC*, 1997.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[RAD78] Ron Rivest, Leonard Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–180, 1978.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.

A More Definitions

A.1 Obfuscation

Definition A.1 (Indistinguishability Obfuscator ($i\mathcal{O}$))**.** A uniform p.p.t. machine $i\mathcal{O}$ is called an *indistinguishability obfuscator* for a circuit class $\{\mathcal{C}_\lambda\}$ if the following conditions are satisfied:

- For all security parameters $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, for all inputs x, we have that
$$\Pr[C'(x) = C(x) : C' \leftarrow i\mathcal{O}(\lambda, C)] = 1$$
- For any (not necessarily uniform) PPT distinguisher D, there exists a negligible function α such that the following holds: For all security parameters $\lambda \in \mathbb{N}$, for all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$, we have that if $C_0(x) = C_1(x)$ for all inputs x, then
$$\left| \Pr\left[D(i\mathcal{O}(\lambda, C_0)) = 1\right] - \Pr\left[D(i\mathcal{O}(\lambda, C_1)) = 1\right] \right| \leq \alpha(\lambda)$$

The work of Garg et al. [GGH+13] presents a candidate construction for indistinguishability obfuscation based on a new assumption about multilinear maps.

A.2 Statistically Simulation Sound Non-Interactive Zero Knowledge (NIZK)

Another primitive introduced by [GGH+13], which we will use, is statistically simulation sound NIZK. It was shown that such NIZKs can be constructed from standard NIZKs and a commitment scheme.

Definition A.2. A statistically simulation-sound non-interactive zero-knowledge proof (K, P, V) for a relation R has the following properties:

Perfect completeness. A proof system is complete if an honest prover with a valid witness can convince an honest verifier. Formally we have

$$\Pr\left[\sigma \leftarrow K(1^\lambda) : \exists(x,\pi) : x \notin L : V(\sigma, x, \pi) = 1\right] = 1.$$

Statistical soundness. A proof system is sound if it is infeasible to convince an honest verifier when the statement is false. For all (even unbounded) adversaries $\mathcal{A}$ we have

$$\Pr\left[\sigma \leftarrow K(1^\lambda); (x,\pi) \leftarrow \mathcal{A}(\sigma) : V(\sigma, x, \pi) = 1 : x \notin L\right] = \mathsf{negl}(\lambda).$$

Computational zero-knowledge [FLS99]. A proof system is computational zero-knowledge if the proofs do not reveal any information about the witnesses to a bounded adversary. We say a non-interactive proof (K, P, V) is computational zero-knowledge if there exists a polynomial time simulator $S = (S_1, S_2)$, where S_1 returns a simulated common reference string σ together with a simulation trapdoor τ that enables S_2 to simulate proofs without access to the witness. For all non-uniform polynomial time adversaries $\mathcal{A}$ we have for all $x \in L$

$$\Pr\left[\mathcal{A}(x,\sigma,\pi) = 1 \ : \ \begin{array}{c}\sigma \leftarrow K(1^\lambda)\\ \pi \leftarrow P(\sigma, x, w)\end{array}\right] \approx \Pr\left[\mathcal{A}(x,\sigma,\pi) = 1 \ : \ \begin{array}{c}(\sigma,\tau) \leftarrow S_1(1^\lambda, x)\\ \pi \leftarrow S_2(\sigma,\tau,x)\end{array}\right].$$

Statistical Simulation-Soundness (SSS). A proof system is said to be statistically simulation sound if it is infeasible to convince an honest verifier of a false statement even when the adversary itself is provided with a simulated proof. For all statements x and all (even unbounded) adversaries $\mathcal{A}$ we have

$$\Pr\left[(\sigma,\tau) \leftarrow S_1(1^\lambda, x); \pi \leftarrow S_2(\sigma,\tau,x) : \exists(x',\pi') : x' \neq x : V(\sigma, x', \pi') = 1 : x' \notin L\right] = \mathsf{negl}(\lambda).$$

B Overview of One-Time GRAM in the Literature

We briefly describe a high-level framework for construction one-time garbled RAM as defined in [LO13a, GHL+14]. The original instantiation of [LO13a] had a subtle bug which was later fixed by [GHL+14]. See [GHL+14] for full details. Our main focus here is on the syntax of the construction and explaining why it satisfies our efficiency requirements such as bit-wise compactness. We begin by recalling the RAM model of computation.

B.1 Random Access Machines

A RAM program can be represented by a *CPU-Step Circuit*

$$\mathsf{CPU}(s, b^{\mathsf{read}}) = (s', i^{\mathsf{read}}, i^{\mathsf{write}}, b^{\mathsf{write}}),$$

which also has access to external memory. The RAM program is executed by repeatedly taking as input the current state s and data b^{read} and applying the CPU-step function from above. This step determines the state for the next step s', some data b^{write} to write to memory location i^{write}, and another memory location i^{read} from which to read the data for the next step. The initial s of the computation consists of the input, and the computation concludes when it reaches a designated "halt" s, at which point the output is found in the external memory, starting from position i^{read}. The initial memory content is often taken to be empty, but when we consider repeated computations with persistent memory then the memory content is assumed to persist from the previous execution.

B.2 The One-Time GRAM Construction

The main idea for how to garble a RAM program P is to just use Yao garbled circuits to garble t copies of the CPU step circuit as described above. The state of the computation s remains garbled from one circuit to the next. The main difficulty is how to allow the computation to access (read/write) values in memory. To do so, we define a "garbled memory" which contains a secret keys $\mathsf{sk}_{i,j,b}$ for each memory location i, where b corresponds to the bit that should be in that location and j corresponds to the CPU step in which that bit was written (or 0 if it was initialized that way). Each garbled CPU step circuit also outputs (in the clear) the locations $i^{\mathsf{read}}, i^{\mathsf{write}}$ that it wants to read/write as well as:

- A secret key $\mathsf{sk}^{\mathsf{write}} = \mathsf{sk}_{i^{\mathsf{write}},j,b^{\mathsf{write}}}$ where b^{write} is the bit being written to in location i^{write} and j is the current CPU step index.
- Two ciphertexts c_0, c_1 that encrypt the labels corresponding to bit 0 and 1 respectively for the wire corresponding to the bit b^{read} in the next garbled circuit. The encryptions are created in such a way that c_b can only be decrypted by $\mathsf{sk}_{i,j,b}$ where j is the time period in which location i was last written to.

Originally, the garbled memory consists of secret keys $\mathsf{sk}_{i,0,D[i]}$ where $D[i]$ is the i'th bit in the persistent data D, or we set $D[i] = 0$ in the case where we have no persistent data and just want to initialize the memory to all 0s. The evaluator starts evaluating the garbled CPU step circuits one-at-a-time. After each evaluation, it gets our the values $i^{\mathsf{read}}, i^{\mathsf{write}}, \mathsf{sk}^{\mathsf{write}}, c_0, c_1$. It writes $\mathsf{sk}^{\mathsf{write}}$ to the location i^{write} of garbled memory and reads $\mathsf{sk}^{\mathsf{read}}$ from location i^{read}. It attempts to decrypt both ciphertexts c_0, c_1 with $\mathsf{sk}^{\mathsf{read}}$ and, depending on whichever one decrypts correctly, it uses the corresponding decrypted message as the label of the bit b^{read} for the next garbled circuit.

Loose Ends. The above already described the main idea, but there are several loose ends in the above description:

- The evaluator learns all of the locations $i^{\mathsf{read}}, i^{\mathsf{write}}$ being accessed by the program, which can reveal sensitive information about the program execition. This can be fixed by using an *oblivious RAM* scheme to compile the computation into one where these locations do not reveal any private information.
- Each garbled CPU circuit, when reading location i^{read} needs to know the index j in which that location was last written to. There are generic ways to convert any program execution into one that makes this easy, and some ORAM constructions already have this type of "predictably-timed writes" property.

The original scheme from [LO13a] has one master secret key that was hard-coded in each garbled CPU step circuit and could be used to generate the values $c_0, c_1, \mathsf{sk}^{\mathsf{write}}$. This lead to a subtle

circularity problem as outlined in [GHL+14], but this can be fixed using *identity-based encryption*, as shown in [GHL+14, GHRW14]. Note that [GHL+14, LO14] also present an alternate approach which does not require identity-based encryption but loses in efficiency, and therefore we do not consider it in this work.

Efficiency. The main take-away from the above description is the following: to create a garbled one-time RAM program with run-time t, we need to create t garbled copies of (an augmented version of) the CPU step circuit under Yao's garbled circuit construction. Each such circuit is of some small size $\mathsf{poly}(\lambda)$. The circuits aren't completely independent - the labels of the output wires in one circuit must match the labels of the input wires of the next circuit. However, in Yao's garbled circuits, we can choose the wire labels pseduorandomly via a PRF of the wire identifier. Therefore, we can compute each of the t garbled circuits individually in time $\mathsf{poly}(\lambda)$ without computing all of the other circuits. In particular, there is a short circuit of size $\mathsf{poly}(\lambda)$ that outputs the ith bit of the one-time garbled program $\tilde{P}$ by only computing a single garbled circuit and outputting 1 bit of it.

C Reusable Garbled Circuits with Output-Size Independence

We show that any construction of reusable garbled circuits satisfying *simulation-based* security must have have garbled inputs of length at least as large as the outputs of the circuit being garbled. In other words, if we garble a circuit with short inputs and huge outputs, the length of the garbled input will unfortunately need to be huge. Note that this is in contrast to *standard* (non-reusable) garbled circuits, for which the above does not hold. In our work, we also show that the above does not hold for reusable garbled circuits if we give up on simulation security, and instead only consider an *indistinguishability*-based security. The impossibility argument follows the "incompressibility analysis" used in [AGVW13]. On a high level, a simulator that can simulate many garbled inputs of a reusable garbled circuit given many outputs has found a way to "compress" the outputs: given the garbled inputs and the garbled circuit one can recover all the outputs. If the outputs are pseduorandom (e.g., the outputs of a pseudoranom generator) then this should be impossible.

Theorem C.1. Let $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ be any reusable garbled-circuit scheme with simulation security. Then for any polynomial $m = m(\lambda)$ there is a family of circuits $C_\lambda : \lambda \to m(\lambda)$ for which the garbled inputs must be of length at least m: i.e., for $x \in \{0,1\}^\lambda$, $(\tilde{C}, k) \leftarrow \mathsf{GC.circ}(1^\lambda, C_\lambda), \tilde{x} \leftarrow \mathsf{GC.inp}(x, k)$, we have $|\tilde{x}| \geq m$. [11]

Proof. For any polynomial $m = m(\lambda)$, let $\mathsf{prg} : \{0,1\}^\lambda \to \{0,1\}^m$ be a pseudo-random generator (PRG), let $C_{\mathsf{prg},\lambda}$ be a circuit family computing the function prg. Assume that the theorem does not hold so there is some $\ell = \ell(\lambda) < m$ such that the garbled inputs to the garbled $C_{\mathsf{prg},\lambda}$ are of size ℓ. Let $p = p(\lambda)$ be some bound on the size of $\tilde{C}_{\mathsf{prg},\lambda}$ given by $(\tilde{C}_{\mathsf{prg},\lambda}, k) \leftarrow \mathsf{GC.circ}(1^\lambda, C_{\mathsf{prg},\lambda})$. Let $q = q(\lambda)$ be such that $q \cdot m > q \cdot \ell + p + \lambda$. Let Sim be the simulator for GC.

We construct a distinguisher Dist for the PRG. The distinguisher gets q values $y_1, \ldots, y_q$ which are either all random or all pseudo-random. $\mathsf{Dist}(1^\lambda, y_1, \ldots, y_q)$ runs $(\tilde{C}'_{\mathsf{prg},\lambda}, \tilde{x}'_1, \ldots, \tilde{x}'_q) \leftarrow \mathsf{Sim}(1^\lambda, C_{\mathsf{prg}}, y_1, \ldots, y_q)$. It tests if $\mathsf{GC.eval}(\tilde{C}'_{\mathsf{prg},\lambda}, \tilde{x}'_i) = y_i$ for all $i \in [q]$ and, if so, it outputs 1 else it outputs 0.

We claim that if $y_1, \ldots, y_q$ are pseudorandom with $y_i = \mathsf{prg}(x_i)$ for some $x_i \in \{0,1\}^\lambda$, then $\mathsf{Dist}(y_1, \ldots, y_q) = 1$ with probability $1 - \mathsf{negl}(\lambda)$. This is because, by the security of the simulation, we have

$$(\tilde{C}'_{\mathsf{prg},\lambda}, \tilde{x}'_1, \ldots, \tilde{x}'_q) \overset{\text{comp}}{\approx} (\tilde{C}_{\mathsf{prg},\lambda}, \tilde{x}_1, \ldots, \tilde{x}_q)$$

[11] For simplicity, we assume that for a fixed C_λ, the sizes of $\tilde{C}, \tilde{x}$ are also fixed and do not vary.

where $(\tilde{C}_{\mathsf{prg},\lambda}, k) \leftarrow \mathsf{GC.circ}(1^\lambda, C_{\mathsf{prg},\lambda})$ is the real garbled circuit and $\tilde{x}_i \leftarrow \mathsf{GC.inp}(x_i, k)$ are the real garbled inputs. By the correctness of the garbled circuit we must have $\mathsf{GC.eval}(\tilde{C}_{\mathsf{prg},\lambda}, \tilde{x}_i) = y_i$ and therefore we must also have $\mathsf{GC.eval}(\tilde{C}'_{\mathsf{prg},\lambda}, \tilde{x}'_i) = y_i$ with overwhelming probability.

On the other hand, if $\bar{y} = (y_1, \ldots, y_q)$ is random then $\mathsf{Dist}(y_1, \ldots, y_q) = 1$ with probability $\mathsf{negl}(\lambda)$. This is because there are 2^{mq} possible values of $\bar{y}$ but only $2^{p+q\ell}$ possible values of $(\tilde{C}'_{\mathsf{prg},\lambda}, \tilde{x}'_1, \ldots, \tilde{x}'_q)$ of the right size. Therefore there is only a $2^{mq-(p+q\ell)} = 2^{-\lambda}$ fraction of values $\bar{y}$ that the simulator can "explain" with some $(\tilde{C}'_{\mathsf{prg},\lambda}, \tilde{x}'_1, \ldots, \tilde{x}'_q)$.

Together, this proves the theorem.

□