Fully Succinct Garbled RAM

Ran Canetti* Justin Holmgren†

April 25, 2015

Abstract

We construct the first fully succinct garbling scheme for RAM programs, assuming the existence of indistinguishability obfuscation for circuits and one-way functions. That is, the size, space requirements, and runtime of the garbled program are the same as those of the input program, up to poly-logarithmic factors and a polynomial in the security parameter. The scheme can be used to construct indistinguishability obfuscators for RAM programs with comparable efficiency, at the price of requiring sub-exponential security of the underlying primitives.

The scheme builds on the recent schemes of Koppula-Lewko-Waters and Canetti-Holmgren-Jain-Vaikuntanathan [STOC 15]. A key technical challenge here is how to combine the fixed-prefix technique of KLW, which was developed for deterministic programs, with randomized Oblivious RAM techniques. To overcome that, we develop a method for arguing about the indistinguishability of two obfuscated randomized programs that use correlated randomness. Along the way, we also define and construct garbling schemes that offer only partial protection. These may be of independent interest.

*Tel-Aviv University and Boston University, canetti@bu.edu. Supported by the Check Point Institute for Information Security, ISF grant 1523/14, and NSF Frontier CNS1413920 and 1218461 grants.

†MIT, holmgren@mit.edu. Supported by NSF Frontier CNS1413920

1 Introduction

A garbling scheme $\mathcal{G}$ converts programs and input values into "opaque" constructs that reveal nothing but the corresponding output values. That is, $\mathcal{G}$ turns a program M into a garbled program $\tilde{M}$ and, separately, turns a value x into a garbled input $\tilde{x}$, with the guarantee that $\tilde{M}(\tilde{x}) = M(x)$ and in addition the pair $(\tilde{M}, \tilde{x})$ reveals nothing but $M(x)$. Originally conceived by Yao [Yao82], garbling schemes are a pillar of cryptographic protocol design, with numerous applications such as secure two-party and multiparty computation protocols, verifiable delegation schemes, randomized encoding schemes, one time programs, and functional encryption.

A drawback of Yao's original construction is that the size and runtime of the garbled program are proportional to the circuit representation of the input program. This holds even if the plaintext program is represented more succinctly, say as a Turing machine or a RAM program. (Essentially, one has to first translate the plaintext program to a circuit, and then apply Yao's garbling method in a gate by gate manner.) This drawback becomes especially significant in situations where the input x is much larger than the program's size or runtime — as in, say, keyword search in a large-but-sorted database — or when the runtime of the plaintext program varies from input to input.

Noticing this drawback, Goldwasser Kalai et al. [GKP+13] construct a garbling scheme for *Turing machines,* namely a scheme where the size, runtime and space requirements of the garbled program are proportional to those of the Turing machine representation of the plaintext program. To do that, they make strong *extractability* assumptions. Namely, they postulate existence of an efficient algorithm for extracting secrets from a certain class of adversaries.

Noticing the same drawback, Lu and Ostrovsky, and later Gentry Halevi et al. and Garg Lu et al. [LO13, GHL+14, GLOS15], construct garbling schemes for RAM programs, where the runtime of the garbled program is proportional only to the runtime of the plaintext program on that input. In [GLOS15] this is done assuming only one way functions. Still, the *size* of the garbled program is proportional to the *runtime* of the plaintext program.

Bitansky Garg et al. and Canetti Holmgren et al. construct a *semi-succinct* garbling scheme for RAM programs, assuming non-succinct Indistinguishability Obfuscation (IO) and injective one way functions [BGL+15, CHJV15]. That is, they construct garbling schemes where the space and runtime of the garbled program are proportional to the space and runtime of the plaintext program, and where the size of the garbled program is proportional to the *space* complexity of the plaintext program. For this they assume existence of non-succinct IO schemes, i.e. schemes where the complexity of the obfuscated program is polynomial in the size of the circuit representation of the plaintext program. (Indeed, current candidate indistinguishability obfuscators are such [GGH+13, BGK+14, Zim14, AB15].) We note that, although the overall parameters of these two schemes are roughly comparable, the underlying techniques are quite different.

Koppula, Lewko and Waters [KLW15] devise a *fully succinct* garbling scheme for Turing machines from non-succinct IO and one way functions, using techniques that extend those of [CHJV15]. That is, in their garbling scheme the runtime, space and description size of the garbled program are proportional to those of the Turing machine representation of the plaintext program. This leaves open the following natural question:

Do there exist fully succinct garbling schemes for RAM programs? If so, under what assumptions?

Any advancement on this question directly applies to the many applications of succinct garbling mentioned in these works, including delegation of computation, functional encryption and others.

From succinct garbling to succinct obfuscation. In [BGL+15, CHJV15] it is also shown how to turn a garbling scheme into a full-fledged program obfuscation scheme with comparable efficiency and succinctness properties, at the price of making stronger assumptions on the underlying cryptographic building blocks. That is, given non-succinct IO (namely IO for circuits), one-way functions, and a garbling scheme $\mathcal{G}$, they construct an IO scheme $\mathcal{O}$ with similar efficiency and size overhead as that for $\mathcal{G}$. The security of $\mathcal{O}$ loses a factor of D, where D is the size of the domain of inputs to the plaintext program. Using this transformation, and assuming sub-exponential one-way functions and IO for circuits, [BGL+15, CHJV15, KLW15] show a fully succinct IO scheme for Turing machines and semi-succinct IO scheme for RAM machines. However:

Is there a fully succinct IO scheme for RAM programs? If so, under what assumptions?

We note that, due to the exponential degradation in security in that transform, the security parameter needs to grow linearly with $\log D$. The size of the obfuscated program thus grows polynomially in the length of input to the plaintext program. We only know how to get below this bound under significantly stronger assumptions on the underlying obfuscation scheme [BCP14, IPS15].

1.1 Our contribution

We answer both questions. Given an IO scheme for circuits and one way functions we construct a fully succinct garbling scheme for RAM programs. That is, the runtime, space, and size of the garbled program are the same as those of the plaintext program, up to polylogarithmic factors and a polynomial in the security parameter. The security of the scheme degrades polynomially with the runtime of the plaintext program. Assuming quasipolynomial security of the underlying primitives, the scheme guarantees full security even for programs with arbitrary polynomial runtime.

Furthermore, similarly to the schemes of [CHJV15, BGL+15, KLW15], we note that our garbling scheme supports persistent data: multiple RAM machines $M_1, \ldots, M_\ell$ can be garbled such that machine M_i acts on the memory configuration left by M_{i-1}. For example, each machine may execute a database query, modifying the database and returning some small result.

Using the transformation of [BGL+15, CHJV15], and assuming sub-exponential security of the underlying primitives, we obtain a fully succinct IO scheme for RAM programs.

Our Techniques. While our result may come across as natural and expected given the results of [KLW15] and [CHJV15], obtaining it does require new ideas and significant work. Indeed, a naive attempt to extend the techniques of [KLW15] to RAM programs immediately encounters the following problem: The [KLW15] technique applies only to *deterministic* programs where the memory access pattern is fixed and independent of the inputs. In contrast, hiding the memory access pattern in a RAM program in an efficiency-preserving way requires the memory access pattern to be randomized. Indeed, Oblivious RAM schemes [GO96] are inherently randomized. Furthermore, the security guarantees provided by Oblivious RAM (ORAM) schemes hold only when the internal random choices of the scheme are hidden from the adversary. In our case these internal random choices are encapsulated in a succinct program that is only protected by indistinguishability obfuscation.

A second look reveals the following basic discrepancy between the [KLW15] technique and that of [CHJV15]. In both works, security of the garbled program is demonstrated by gradually moving, in a way that's indistinguishable to the adversary, from the real garbled program to a dummy garbled program, where the dummy program has just the result hardwired and is running a fake computation in all steps but the last one. In [CHJV15], the intermediate, hybrid programs start with some number, i, of dummy steps, and then continue the computation from the ith intermediate configuration all the way to the end. To make this technique work with ORAM, [CHJV15] use an ORAM scheme with a strong *forward security* property: Essentially, the addresses accessed before time i must appear independent of the underlying access pattern, even given the scheme's internal state at time $i+1$.

In contrast, [KLW15] move from the real garbled program to the dummy one via intermediate programs that perform the computation from the beginning until some step, i. From then on, the intermediate program performs the dummy computation and in the end it outputs its hardwired value. This reversal of the order of steps in the intermediate programs is the key idea that allows the size of their garbled program to not depend on the space requirements of the plaintext program. However, the new method seems incompatible with ORAM techniques: Indeed, the natural way to extend the [KLW15] argument to this case would be to argue that the program's memory access pattern is random even given the program's state at step $i-1$. But this does not hold, since all the steps of the computation up to the transition point i are executed, including the internal random choices of whatever ORAM scheme is in use.

Our first step towards getting around this difficulty is to identify the following property of ORAM schemes. Recall that an ORAM scheme translates the memory access requests made by the underlying program

to randomized locations in the actual external memory. We say that an ORAM scheme has **localized randomness** if the random variable describing the physical location of the memory cell accessed by the plaintext program at a certain step of the computation depends only on a relatively small portion of the entire random input of the ORAM scheme. Furthermore, we require that the location of this portion depends only on the last step in which this memory cell was accessed, which in of itself is a deterministic function of the underlying program. To the best of our knowledge, this property of Path ORAMs has not been utilized in previous work, but we observe that the ORAM of [CP13] has localized randomness. (In fact, we conjecture that other schemes do as well, or can be slightly modified to be so.) Now, given an ORAM scheme with localized randomness, we "puncture" the scheme at exactly the points that are necessary for making the external memory access locations at step i appear random even given the punctured program state at step $i-1$. Furthermore, we can perform this puncturing with minimal overhead in terms of the size of the obfuscated program.

More concretely, we proceed in two main steps. (The actual construction goes through a number of smaller steps, for sake of modularity and clarity.) We first build a "fixed-address" garbler which guarantees that the garbled versions of two machines M_0 and M_1 with inputs x_0 and x_1 are indistinguishable as long they access the same sequence of addresses. We believe that this property is of independent interest. In the second step we use an ORAM scheme with localized randomness to obtain full garbling. Below we provide more detail on these two steps.

1.1.1 Fixed Address Garbling

As an intermediary step towards a fully succinct garbling scheme for RAM programs, we define and obtain the following weaker security property for garbling schemes. We say that a garbling scheme is a *fixed-address garbler* if for any two same-size deterministic programs M_0 and M_1 and same-length input values x_0 and x_1, such that (a) $M_0(x_0) = M_1(x_1)$ and (b) The sequence of memory addresses accessed by M_0 when run on x_0 is identical to the sequence of memory addresses accessed by M_1 when run on x_1, it holds that $(\tilde{M}_0, \tilde{x}_0) \approx (\tilde{M}_1, \tilde{x}_1)$. (Here $\tilde{M}$ and $\tilde{x}$ are the garbled versions of M and x, respectively.) Furthermore, we require that the sequence of addresses accessed by $\tilde{M}$ on input $\tilde{x}$ is identical to the sequence of addresses accessed by M on input x.

Fixed-address garbling appears to be a natural notion. Indeed, in a way it is comparable to Indistinguishability Obfuscation: As long as the "externally observable behavior" of two programs is the same, their garbled/obfuscated versions are indistinguishable. Furthermore, the fact that the access pattern is preserved provides potential efficiency and practical applicability gains that are not possible in the context of fully secure and succinct garbling of RAM programs, since in the latter the access pattern is inherently randomized. For instance, the garbled machine necessarily has the same fine-grain cache performance as the original one. In contract, ORAM-based techniques need to resort to coarse-grain cache or other work-arounds which impact cache performance.

We construct a fully succinct fixed-address garbling scheme. As a preliminary step, we construct a garbling scheme that is fixed-address, except that $(\tilde{M}_0, \tilde{x}_0) \approx (\tilde{M}_1, \tilde{x}_1)$ only when the two computations have the exact same memory access pattern, including the contents of the memory cells accessed. (We call such schemes *fixed-memory* garbling schemes.) Here our technique follows the steps of the [KLW15] machine-hiding encoding scheme. In particular we use the same underlying primitives, namely positional accumulators, cryptographic iterators, and splittable signatures. (We somewhat simplify their interfaces.) We note however that the [KLW15] construction cannot be used in a "black box" way and needs to be redone in the RAM model.

We then move from fixed-memory garbling to fixed-address garbling. Similarly to the move in [KLW15] from machine-hiding encoding to garbling, this step requires encrypting the memory contents in an IO-friendly scheme. We stress however that our situation is different: Indeed, in their oblivious Turing machine model the memory access pattern contains no information. In contrast, as argued in more detail below, in our case the access pattern can in of itself contain information that is hard to compress in a security-preserving manner. Therefore, the way we argue about the security of the scheme must change accordingly.

Concretely, to garble M we transform it to a program M' which interleaves *two* executions of M, on

two parallel tracks 'A' and 'B' of memory. Whenever M would access a memory address, M' accesses the corresponding address in both tracks 'A' and 'B'. At each point in time, tracks 'A' and 'B' both store memory contents corresponding to an execution of M. We then apply the fixed-memory garbling scheme to M'. Let $\tilde{M}'$ denote the resulting program.

To argue fixed-address security, consider two programs M_0 and M_1 and input values x_0 and x_1 that satisfy the fixed-address requirements. To show that $(\tilde{M}'_0, \tilde{x}_0) \approx (\tilde{M}'_1, \tilde{x}_1)$, we consider an intermediate hybrid in which M'_0 is replaced by a new machine M_{01} which now executes M_0 on track 'A' but M_1 on track 'B'. Once we get here, a symmetric argument shows how to replace M_{01} by the machine M'_1 which executes M_1 in both tracks 'A' and 'B'. The only remaining part, switching the encoded input from x_0 to x_1, does not involve any more technical legwork.

One might naturally wonder why the double execution of M_0 is necessary; why cannot one just use a different hybrid argument wherein the execution of M_0 is changed to a dummy computation, and then the dummy computation is changed to an execution of M_1? The reason is that the dummy computation may actually contain less information than either M_0 or M_1. Indeed, the memory access pattern itself can contain infromation that is not efficiently reproducible by a succinct dummy machine. This argument is similar to the incompressibility argument of Hubáček and Wichs [HW14] which lower bounds the communication complexity of secure computation with long outputs.

1.1.2 Full Garbling

Our final and main result is a construction showing that the existence of a succinct fixed-address garbler implies a succinct fully secure garbler for RAM machines. For a fully secure garbler, the garblings of RAM machines M_0 and M_1 on respective inputs x_0 and x_1 must be indistinguishable if $M_0(x_0) = M_1(x_1)$, and the runtime and space requirement of M_0 on x_0 is the same as the runtime and space requirement of M_1 on x_1. Our construction is fully general; it does not use any special properties of the fixed-address garbler, not even the address-preserving property which we explicitly highlighted above.

If we didn't care about preserving the RAM efficiency of the input program, then we could garble a machine M on input x by first applying the trivial "brute-force" ORAM which accesses every address in memory per input access, and then applying the fixed-address garbler. This would be secure because the brute-force ORAM transforms all machines to have identical access pattern. In contrast, we are interested in ORAM schemes with only only polylogarithmic overhead; here the memory access pattern is inherently randomized, and the hiding guarantees regarding the access patterns are *distributional*.

A first step towards applying a fixed-address garbler is to make the ORAM scheme deterministic by generating its randomness by applying a puncturable PRF to the program's input. Still, it is not clear how to argue security of the scheme. For this purpose, we use the *localized randomness* property sketched above and described in more detail here. Localized randomness requires a particularly structured relationship between the random tape R of an ORAM and the addresses $\mathbf{a}_1, \ldots, \mathbf{a}_t$ that it accesses. Specifically, it requires that (for given underlying memory operations $\mathsf{op}_1, \ldots, \mathsf{op}_t$), each $\mathbf{a}_i$ (which is itself a sequence of addresses $a_{i,1}, \ldots, a_{i,\eta}$ for some small η) depends on a small subset D_i of the bits of R. Furthermore, D_i must be efficiently computable and for each $i \neq j$, D_i and D_j must be disjoint. There must also be some fixed algorithm $\mathsf{OSample}$ such that regardless of $\mathsf{op}_1, \ldots, \mathsf{op}_t$, $\mathsf{OSample}(i)$ has the same distribution as $\mathbf{a}_i$. A simple analysis in Appendix A shows that the ORAM of Chung and Pass [CP13, SCSL11] has this property.

To analyze the composition of a fixed-address garbler with a localized-randomness ORAM, we adapt the punctured programming technique of [SW14]. To simulate a garbled program whose output is y and runs in time T, apply a fixed-address garbler to the dummy program that for each i from 1 to T, accesses the addresses given by $\mathsf{OSample}(i; F(i))$ for some puncturable PRF F, and output the resulting garbled program. Now to prove that this simulation is indistinguishable from the garbled version of a given machine M, we change each $\mathbf{a}_i$ to $\mathsf{OSample}(i; F(i))$ in a sequence of indistinguishable hybrids.

This argument is reminiscent of the proof of security for the [CLTV15] construction of a probabilistic i$\mathcal{O}$ (PIO) obfuscator, with the complication that $\mathbf{a}_1$ through $\mathbf{a}_t$ are generated adaptively. This complication is handled by switching the $\mathbf{a}_i$'s in reverse order – starting with $\mathbf{a}_t$ and ending with $\mathbf{a}_1$. Here it is crucial to

note that, despite the adaptivity, $\mathbf{a}_1$ through $\mathbf{a}_t$ are mutually independent random variables by the localized randomness property of the ORAM scheme.

To switch $\mathbf{a}_i$ to $\mathsf{OSample}(i; F(i))$, we first hard-code $\mathbf{a}_i$, and then puncture the ORAM's PRF on exactly the points which determine $\mathbf{a}_i$. ORAM locality implies that this set is *small* and that the puncturing does not affect any $\mathbf{a}_j$ for $j \neq i$. We indistinguishably replace $\mathbf{a}_i$ with $\mathsf{OSample}(i)$, and then with $\mathsf{OSample}(i; F(i))$. Finally we remove the hard-codings and unpuncture all the PRFs.

1.2 Roadmap

As mentioned, we build up our main construction in four stages, at each stage strengthening the security properties. In the first two stages, we directly apply the techniques of [KLW15] to produce a very weak garbling scheme for RAM machines. For ease of exposition, we separate this into two parts. In Section 3, we give a garbler which only guarantees indistinguishability of the garbled programs as long as the entire execution transcripts of the two plaintext machines look identical; that is, if they specify the same sequence of internal local states, same addresses accessed, and same values written to memory. We call such schemes *fixed transcript garblers.* In Section 4, we upgrade this garbling scheme to a *fixed-memory garbler,* which no longer needs the machines to have the same internal local states.

Our main technical contributions are the construction of a *fixed-address garbler* in Section 5, and its combination with a local ORAM in Section 6 to build a full RAM garbler.

In Appendix A, we describe the ORAM of Chung and Pass [CP13], and explain why it has the desired locality properties.

2 Preliminaries

2.1 RAM Machines

In this work, a RAM program (or, machine) M is defined as a tuple (Σ, Q, Y, C), where:

- Σ is a finite set, which is the possible contents of a memory cell. We assume that Σ contains an "empty" symbol ϵ. Say, $\Sigma = \{0, 1, \epsilon\}$.
- Q is the set of all possible "local states" of M, containing some initial state q_0. (We think of Q as a set that grows polynomially as a function of the security parameter. That is, a state $q \in Q$ can encode cryptographic keys, as well as "local memory" of size that is bounded by some fixed polynomial in the security parameter.)
- Y is the output space of M.
- C is a circuit implementing a transition function which maps $Q \times \Sigma \to (Q \times \mathbb{N} \times \{\mathsf{Read}, \mathsf{Write}\} \times \Sigma) \cup Y$. (That is, C takes the current state and the value returned by the memory access function, and returns a new state, a memory address, a read/write instruction, and a value to be written in case of a write. We call the tuple (*memory address, instruction, value*) a **memory access tuple.**)

2.1.1 Evaluation

To define an evaluation of a RAM machine we first need to define memory configurations. Formally, a memory configuration is a function $s : \mathbb{N} \to \Sigma$. More concretely, we think of a memory configuration as a data structure s that given an index i returns $s[i]$. Naturally, s can be implemented efficiently with size that grows linearly with the number of non-empty memory locations, and with access time that grows logarithmically in that number. Let $|s|$ denote the number of non-empty locations in s.

The evaluation of a RAM machine $M = (\Sigma, Q, Y, C)$ is defined as the following function mapping initial memory configurations to either $\perp$ or an output $y \in Y$. Arbitrarily define $a_0 = 0$, and for $i > 0$, iteratively

compute $(q_i, a_i, \mathsf{op}_i, v_i) \leftarrow C(q_{i-1}, s_{i-1}(a_{i-1}))$, and define

$$s_i(a) = \begin{cases} v_i & \text{if } a = a_i \text{ and } \mathsf{op}_i = \mathsf{Write} \\ s_{i-1}(a) & \text{otherwise} \end{cases}$$

If the non-empty portion of s_0 is a prefix of s_0 and contains the string x, and $C(q_{i-1}, s_{i-1}(a_{i-1})) = y \in Y$ for some i, then we say that $M(x) = y$. If there is no such i, we say that $M(x) = \perp$.

2.2 Garbling

Definition 2.1 (Garbling)**.** A garbling scheme for RAM progams is a pair of algorithms $(\mathsf{Garble}, \mathsf{Eval})$, such that:

- Garble takes as input a RAM machine M, a memory configuration x, a time bound T, and a space usage S, and then produces as output a garbled RAM machine $\tilde{M}$ and a garbled memory configuration $\tilde{x}$.
- Eval takes a garbled RAM machine $\tilde{M}$ and a garbled memory configuration $\tilde{x}$ and outputs a value y.
- Garble can be decomposed into three algorithms: $(\mathsf{KeyGen}, \mathsf{GbPrg}, \mathsf{GbInp})$ such that $\mathsf{Garble}(M, x, T, S)$ first runs $K \leftarrow \mathsf{KeyGen}(1^\lambda, S)$, and then computes $\tilde{M} \leftarrow \mathsf{GbPrg}(K, M, T)$ and $\tilde{x} \leftarrow \mathsf{GbInp}(K, x)$.

We are interested in garbling schemes which are *correct*, *efficient*, and *secure*.

Definition 2.2 (Correctness)**.** A garbling scheme $(\mathsf{Garble}, \mathsf{Eval})$ is said to be correct if for all RAM programs M and inputs x such that $M(x)$ runs in time less than T and space less than S, we have $\mathsf{Eval}(\mathsf{Garble}(M, x, T, S)) = M(x)$ with high probability.

Definition 2.3 (Garble Efficiency)**.** Garble is said to be efficient if KeyGen, GbPrg, and GbInp are all probablistic polynomial-time algorithms. In particular, we emphasize that:

- The bounds T and S are encoded in binary, so the time to garble does not significantly depend on either of these quantities.
- GbInp must run in time $\mathrm{poly}(|x|)$, which can be much smaller than the total amount of memory.

Definition 2.4 (Eval Efficiency)**.** Eval is said to be efficient if $\mathsf{Eval}(\mathsf{Garble}(M, x))$ runs in time $\tilde{O}(T_x)$ and space $\tilde{O}(S_x)$, where T_x and S_x are the time and space used by M when executed on x.

Definition 2.5 (Full Security)**.** A garbling scheme $(\mathsf{Garble}, \mathsf{Eval})$ is said to be *secure* if there is an efficient algorithm Sim such that for all RAM programs M and inputs x,

$$\mathsf{Garble}(M, x) \approx \mathsf{Sim}(M(x), T_x, S_x, |M|, |x|).$$

All the garbling schemes we consider are correct and efficient. They have progressively stronger security.

3 Fixed-Transcript Garbling

We first construct a garbling scheme with a very weak security definition. Both the construction and the security proof closely follow the techniques of [KLW15], adapting them to RAM machines.

Definition 3.1. A garbling scheme $(\mathsf{Garble}, \mathsf{Eval})$ is said to be fixed-transcript secure if for all RAM machines $M_0 = (\Sigma, Q, Y, C_0)$ and $M_1 = (\Sigma, Q, Y, C_1)$, we have that $\mathsf{Garble}(M_0, x) \approx \mathsf{Garble}(M_1, x)$ as long as:

- $M_0(x) = M_1(x)$
- $|C_0| = |C_1|$
- The execution transcripts of $M_0(x)$ and $M_1(x)$, including the sequences of all local states and memory access operations, are identical.

3.1 Building Blocks

In addition to indistinguishability obfuscation for circuits, we use the following building blocks. The definitions we give here are equivalent to those in [KLW15], but slightly simplified.

3.1.1 Cryptographic Iterators

Roughly speaking, a cryptographic iterator is a family of collision-resistant hash functions which is i$\mathcal{O}$-friendly when used to authenticate a chain of values. In particular, we think of using a hash function H to hash a chain of values $m_k, \ldots, m_1$ as $H(m_k \| H(m_{k-1} \| \cdots H(m_1 \| 0^\lambda)))$, which we shall denote as $H^k(m_k, \ldots, m_1)$. A cryptographic iterator provides two indistinguishable ways of sampling the hash function H. In addition to "honest" sampling, one can also sample H so that for a sequence of messages $(m_1, \ldots, m_k)$, $H^k(m_k, \ldots, m_1)$ has exactly one pre-image under H.

Below, we give the exact same definition of cryptographic iterators as in [KLW15], only renaming $\mathsf{Setup\text{-}Itr}$ to Setup and renaming $\mathsf{Setup\text{-}Itr\text{-}Enforce}$ to $\mathsf{SetupEnforce}$. Formally, a cryptographic iterator for the message space $\mathcal{M} = \{0,1\}^n$ consists of the following probabilistic polynomial-time algorithms. Setup and $\mathsf{SetupEnforce}$ are randomized algorithms, but $\mathsf{Iterate}$ is deterministic, corresponding to our above discussion of a hash function.

We recall that [KLW15] construct iterators from IO for circuits and puncturable PRFs.

$\mathsf{Setup}(1^\lambda, T) \to \mathsf{PP}, \mathsf{itr}_0$

Setup takes as input the security parameter λ in unary and a binary bound T on the number of iterations. Setup then outputs public parameters PP and an initial iterator value itr_0.

$\mathsf{SetupEnforce}(1^\lambda, T, (m_1, \ldots, m_k)) \to \mathsf{PP}, \mathsf{itr}_0$

$\mathsf{SetupEnforce}$ takes as input the security parameter λ in unary, a binary bound T on the number of iterations, and an arbitrary sequence of messages $m_1, \ldots, m_k$, each in $\{0,1\}^n$ for $k < T$. $\mathsf{SetupEnforce}$ then outputs public parameters PP and an initial iterator value itr_0.

$\mathsf{Iterate}(\mathsf{PP}, \mathsf{itr}_{in}, m) \to \mathsf{itr}_{out}$

$\mathsf{Iterate}$ takes as input public parameters PP, an iterator itr_{in}, and a message $m \in \{0,1\}^n$. $\mathsf{Iterate}$ then outputs a new iterator value itr_{out}. It is stressed that $\mathsf{Iterate}$ is a deterministic operation; that is, given PP, each sequence of messages results in a unique iterator value.

We will recursively define the notation $\mathsf{Iterate}^0(PP, ...) = \mathsf{itr}_0$, and

$$\mathsf{Iterate}^k(\mathsf{PP}, \mathsf{itr}, (m_1, \ldots, m_k)) = \mathsf{Iterate}(\mathsf{PP}, \mathsf{Iterate}^{k-1}(\mathsf{PP}, \mathsf{itr}, (m_1, \ldots, m_{k-1})), m_k).$$

A cryptographic iterator must satisfy the following properties.

Indistinguishability of Setup

For any time bound T and any sequence of messages $m_1, \ldots, m_k$ with $k < T$, it must be the case that

$$\mathsf{Setup}(1^\lambda, T) \approx \mathsf{SetupEnforce}(1^\lambda, T, m_1, \ldots, m_k).$$

Enforcing

Sample $(\mathsf{PP}, \mathsf{itr}_0) \leftarrow \mathsf{SetupEnforce}(1^\lambda, T, (m_1, \ldots, m_k))$.

The enforcement property requires that when $(\mathsf{PP}, \mathsf{itr}_0)$ are sampled as above, $\mathsf{Iterate}(\mathsf{PP}, a, b) = \mathsf{Iterate}^k(\mathsf{PP}, \mathsf{itr}_0, (m_1, \ldots, m_k))$ if and only if $a = \mathsf{Iterate}^{k-1}(\mathsf{PP}, \mathsf{itr}_0, (m_1, \ldots, m_{k-1}))$ and $b = m_k$.

3.1.2 Positional Accumulators

Positional accumulators (PAs) are an i$\mathcal{O}$-friendly version of the well-known Merkle commitments [Mer88]. Merkle commitments (also known as Merkle trees) provide a short computationally-binding commitment of a large database, which can be succinctly and locally opened for a particular address of the database. Merkle trees have many other nice properties. In particular, as one changes the underlying database, the corresponding commitment can be efficiently updated with authentication.

The key additional property of PA's is that this authentication is in some sense "pseudo-information theoretic". More precisely, the public parameters can be (indistinguishably) alternately generated so that for a commitment of specific memory contents $\mathcal{M}^*$ and a specific address addr^*, the only valid opening of address addr^* is the correct one.

More formally, a positional accumulator consists of the following polynomial-time algorithms. SetupAcc and SetupAccEnforceUpdate are randomized, while Update and LocalUpdate are deterministic. For a given memory configuration x, there is a uniquely defined accumulator value ac_x. The procedures Update and LocalUpdate allow for efficient local update and opening. The memory operations supported are either of the form $\mathsf{Read}(\mathsf{addr}_i)$ for some address addr_i, or of the form $\mathsf{Write}(\mathsf{addr}_i \mapsto m_i)$ for some message m_i.

$\mathsf{SetupAcc}(1^\lambda, S) \to \mathsf{PP}, \mathsf{ac}_0, \mathsf{store}_0$

The setup algorithm takes as input the security parameter λ in unary and a bound S (in binary) on the memory addresses accessed. SetupAcc produces as output public parameters PP, an initial accumulator value ac_0, and an initial data store store_0.

This algorithm will be run by the garbler's key generation algorithm. The initial accumulator value will be part of the initial state of the garbled program, and the initial store value will be part of the garbled input. Throughout the execution of a garbled machine, the accumulator value will be a part of the local CPU state, while the data store will be maintained externally by the evaluator.

$\mathsf{Update}(\mathsf{PP}, \mathsf{store}_{in}, \mathsf{op}) \to \mathsf{store}_{out}, \mathsf{aux}$

The prep-update algorithm takes as input the public parameters PP, data store store_{in}[1], and operation op. PrepUpdate then outputs a new data store store_{out} and some auxiliary information aux.

This algorithm will be run by the evaluator of a garbled machine. Informally speaking, aux contains the results of executing op, together with enough information to authenticate these results against a corresponding accumulator value, as well as produce the next accumulator value.

$\mathsf{LocalUpdate}(\mathsf{PP}, \mathsf{ac}_{in}, \mathsf{op}, \mathsf{aux}) \to (m, \mathsf{ac}_{out})$ **or** $\perp$

The local update algorithm takes as inputs the public parameters PP, an accumulator value ac_{in}, a memory operation op, and some auxiliary information aux. LocalUpdate then either outputs a message m and a new accumulator value ac_{out}, or LocalUpdate outputs $\perp$.

This algorithm will be run by the garbled machine itself. Informally speaking, it will be computationally intractable to find a value of aux which induces a non-$\perp$ output of LocalUpdate other than the honestly generated one.

$\mathsf{SetupAccEnforceUpdate}(1^\lambda, S, \mathsf{op}_1, \ldots, \mathsf{op}_k) \to \mathsf{PP}, \mathsf{ac}_0, \mathsf{store}_0$

The alternate setup algorithm additionally take as inputs a sequence of memory operations $\mathsf{op}_1, \ldots, \mathsf{op}_k$ for some integer k. SetupAccEnforceUpdate outputs public parameters PP, an initial accumulator value ac_0, and an initial data store store_0.

This algorithm will be run by the hybrid garblers in the security proof. The difference from the output of SetupAcc is that the output of SetupAccEnforceUpdate satisfies an additional information theoretic "enforcing" property.

A positional accumulator must satisfy the following properties.

[1] Technically for evaluation to be efficient, the input store_{in} should be a *pointer* to a data store

Correctness

Let $\mathsf{op}_0, \ldots, \mathsf{op}_k$ be any arbitrary sequence of memory operations.

We first define the "correct" m_i^* as follows. Say that op_i accesses address addr_i. If no op_j for $j < i$ is a write to addr_i, then m_i^* is ϵ. Otherwise, let j_i be the largest j such that $j < i$ and op_j is a write to addr_i. We define m_i^* such that op_j is of the form $\mathsf{Write}(\mathsf{addr}_i \mapsto m_i^*)$.

Correctness requires that for all $j \in \{0, \ldots, k\}$

$$\Pr\left[m_j = m_j^* : \begin{array}{l} \mathsf{PP}, \mathsf{ac}_0, \mathsf{store}_0 \leftarrow \mathsf{SetupAcc}(1^\lambda, S) \\ \text{For } i = 0, \ldots, k\text{:} \\ \quad \mathsf{store}_{i+1}, \mathsf{aux}_i \leftarrow \mathsf{Update}(\mathsf{PP}, \mathsf{store}_i, \mathsf{op}_i) \\ \quad (m_i, \mathsf{ac}_{i+1}) \leftarrow \mathsf{LocalUpdate}(\mathsf{PP}, \mathsf{ac}_i, \mathsf{op}_i, \mathsf{aux}_i) \end{array}\right] = 1$$

Note we are implicitly requiring that for each i, $\mathsf{LocalUpdate}(\mathsf{PP}, \mathsf{ac}_i, \mathsf{op}_i, \mathsf{aux}_i)$ does not output $\perp$.

Setup Indistinguishability

For any sequence of operations $\mathsf{op}_0, \ldots, \mathsf{op}_k$, any space bound S, and any p.p.t. algorithm $\mathcal{A}$, setup indistinguishability requires that

$$\mathsf{SetupAcc}(1^\lambda, S) \approx \mathsf{SetupAccEnforceUpdate}(1^\lambda, S, \mathsf{op}_0, \ldots, \mathsf{op}_k)$$

Enforcing

Enforcing requires that for all space bounds S, all sequences of operations $\mathsf{op}_1, \ldots, \mathsf{op}_k$, and all aux', we have

$$\Pr\left[v \in \left\{(m_i^*, \mathsf{ac}_{k+1}), \perp\right\} : \begin{array}{l} \mathsf{PP}, \mathsf{ac}_0, \mathsf{store}_0 \leftarrow \mathsf{SetupAccEnforceUpdate}(1^\lambda, S, \mathsf{op}_0, \ldots, \mathsf{op}_k) \\ \text{For } i = 0, \ldots, k-1 \\ \quad \mathsf{store}_{i+1}, \mathsf{aux}_i \leftarrow \mathsf{Update}(\mathsf{PP}, \mathsf{store}_i, \mathsf{op}_i) \\ \quad (m_i, \mathsf{ac}_{i+1}) \leftarrow \mathsf{LocalUpdate}(\mathsf{PP}, \mathsf{ac}_i, \mathsf{op}_i, \mathsf{aux}_i) \\ v \leftarrow \mathsf{LocalUpdate}(\mathsf{PP}, \mathsf{ac}_k, \mathsf{op}_k, \mathsf{aux}') \end{array}\right] = 1$$

Again, we are implicitly requiring that for each $i \in \{0, \ldots, k-1\}$, $\mathsf{LocalUpdate}(\mathsf{PP}, \mathsf{ac}_i, \mathsf{op}_i, \mathsf{aux}_i)$ does not output $\perp$.

Syntactic Differences from [KLW15] Our definition of positional accumulators is syntactically simplified from [KLW15]. Still, the [KLW15] construction satisfies this definition. The main difference is that [KLW15] has separate enforcing setup algorithms for read operations and write operations, as well as having separate update and local-update algorithms.

3.1.3 Splittable Signatures

A splittable signature scheme for a message space $\mathcal{M}$ is a signature scheme whose keys are *constrainable* to certain subsets of $\mathcal{M}$ – namely point sets, the complements of point sets, and the empty set. These punctured keys are required to satisfy indistinguishability and correctness properties similar to the asymmetrically constrained encapsulation of [CHJV15]. Additionally, they must satisfy a "splitting indistinguishability" property.

More formally, a splittable signature scheme syntactically consists of the following polynomial-time algorithms. Setup and Split are randomized algorithms, and Sign and Verify are deterministic.

$\mathsf{Setup}(1^\lambda) \to \mathsf{sk}_\mathcal{M}, \mathsf{vk}_\mathcal{M}$

Setup takes the security parameter λ in unary, and outputs a secret key $\mathsf{sk}_\mathcal{M}$ and a verification key $\mathsf{vk}_\mathcal{M}$ for the whole message space. We will sometimes write the unconstrained keys $\mathsf{sk}_\mathcal{M}$ and $\mathsf{vk}_\mathcal{M}$ as just sk and vk, respectively.

$\mathsf{Split}(\mathsf{sk}_{\mathcal{M}}, m) \to \mathsf{sk}_{\{m\}}, \mathsf{sk}_{\mathcal{M}\setminus\{m\}}, \mathsf{vk}_{\varnothing}, \mathsf{vk}_{\{m\}}, \mathsf{vk}_{\mathcal{M}\setminus\{m\}}$

Split takes as input an unconstrained secret key $\mathsf{sk}_{\mathcal{M}}$ and a message m, and outputs secret keys and verification keys which are constrained on the set $\{m\}$ and its complement $\mathcal{M} \setminus \{m\}$. We note that $\mathsf{sk}_{\{m\}}$ can just be $\mathsf{Sign}(\mathsf{sk}, m)$

$\mathsf{Sign}(\mathsf{sk}_S, m) \to \sigma$

Sign takes a possibly constrained secret key sk_S and a message $m \in S$, and outputs a signature σ.

$\mathsf{Verify}(\mathsf{vk}, m, \sigma) \to 0$ **or** 1

Verify takes a possibly constrained verification key vk, a message m, and a signature σ. Verify outputs 0 or 1. If Verify outputs 1, we say that vk accepts σ as a signature of m; otherwise, we say that vk rejects σ.

A splittable signature scheme must satisfy the following properties.

Correctness

For any message m^*, sample $\mathsf{sk}_{\{m^*\}}$, $\mathsf{sk}_{\mathcal{M}\setminus\{m^*\}}$, $\mathsf{sk}_{\mathcal{M}}$, $\mathsf{vk}_{\varnothing}$, $\mathsf{vk}_{\{m^*\}}$, $\mathsf{vk}_{\mathcal{M}\setminus\{m^*\}}$, and $\mathsf{vk}_{\mathcal{M}}$ as

$$(\mathsf{sk}_{\mathcal{M}}, \mathsf{vk}_{\mathcal{M}}) \leftarrow \mathsf{Setup}(1^\lambda)$$

and

$$(\mathsf{sk}_{\{m^*\}}, \mathsf{sk}_{\mathcal{M}\setminus\{m^*\}}, \mathsf{vk}_{\varnothing}, \mathsf{vk}_{\{m^*\}}, \mathsf{vk}_{\mathcal{M}\setminus\{m^*\}}) \leftarrow \mathsf{Split}(\mathsf{sk}_{\mathcal{M}}, m^*)$$

Correctness requires that with probability 1 over the above sampling:

1. For all $m \in \mathcal{M}$, $\mathsf{Verify}(\mathsf{vk}_{\mathcal{M}}, m, \mathsf{Sign}(\mathsf{sk}_{\mathcal{M}}, m)) = 1$
2. For all sets $S \in \left\{\{m^*\}, \mathcal{M} \setminus \{m^*\}\right\}$, for all $m \in S$, $\mathsf{Sign}(\mathsf{sk}_S, m) = \mathsf{Sign}(\mathsf{sk}_{\mathcal{M}}, m)$. Furthermore, $\mathsf{Verify}(\mathsf{vk}_S, m, \cdot)$ is the same function as $\mathsf{Verify}(\mathsf{vk}_{\mathcal{M}}, m, \cdot)$.
3. For all sets $S \in \left\{\varnothing, \{m^*\}, \mathcal{M} \setminus \{m^*\}, \mathcal{M}\right\}$, for all $m \in \mathcal{M} \setminus S$, and for all σ, $\mathsf{Verify}(\mathsf{vk}_S, m, \sigma) = 0$.

Verification Key Indistinguishability

Sample $\mathsf{sk}_{\{m^*\}}$, $\mathsf{sk}_{\mathcal{M}\setminus\{m^*\}}$, $\mathsf{sk}_{\mathcal{M}}$, $\mathsf{vk}_{\varnothing}$, $\mathsf{vk}_{\{m^*\}}$, $\mathsf{vk}_{\mathcal{M}\setminus\{m^*\}}$, and $\mathsf{vk}_{\mathcal{M}}$ as in the above definition of correctness.

Verification Key Indistinguishability requires that the following indistinguishabilities hold:

1. $\mathsf{vk}_{\varnothing} \approx \mathsf{vk}_{\mathcal{M}}$
2. $\mathsf{sk}_{\{m^*\}}, \mathsf{vk}_{\{m^*\}} \approx \mathsf{sk}_{\{m^*\}}, \mathsf{vk}_{\mathcal{M}}$
3. $\mathsf{sk}_{\mathcal{M}\setminus\{m^*\}}, \mathsf{vk}_{\mathcal{M}\setminus\{m^*\}} \approx \mathsf{sk}_{\mathcal{M}\setminus\{m^*\}}, \mathsf{vk}_{\mathcal{M}}$

Splitting Indistinguishability

Sample $\mathsf{sk}_{\{m^*\}}$, $\mathsf{sk}_{\mathcal{M}\setminus\{m^*\}}$, $\mathsf{vk}_{\{m^*\}}$, and $\mathsf{vk}_{\mathcal{M}\setminus\{m^*\}}$ as in the above definition of correctness. Repeat this sampling, obtaining $\mathsf{sk}'_{\{m^*\}}$, $\mathsf{sk}'_{\mathcal{M}\setminus\{m^*\}}$, $\mathsf{vk}'_{\{m^*\}}$, and $\mathsf{vk}'_{\mathcal{M}\setminus\{m^*\}}$

Splitting indistinguishability requires that

$$\mathsf{sk}_{\{m^*\}}, \mathsf{sk}_{\mathcal{M}\setminus\{m^*\}}, \mathsf{vk}_{\{m^*\}}, \mathsf{vk}_{\mathcal{M}\setminus\{m^*\}} \approx \mathsf{sk}'_{\{m^*\}}, \mathsf{sk}_{\mathcal{M}\setminus\{m^*\}}, \mathsf{vk}'_{\{m^*\}}, \mathsf{vk}_{\mathcal{M}\setminus\{m^*\}}$$

Syntactic Differences from [KLW15] The definition of splittable signatures in [KLW15] is superficially different from ours, but equivalent. Specifically, they do the following differently:

- They give different names for the different types of keys - they omit a subscript of a $\mathcal{M}$ for their "normal" keys, and use a subscript of "one" or "abo" in place of $\{m\}$ and $\mathcal{M} \setminus \{m\}$, respectively.
- Their $\mathsf{sk}_{\{m\}}$ is just defined as $\mathsf{Sign}(\mathsf{sk}_{\mathcal{M}}, m)$, and is thus not an output of Split.
- They generate $\mathsf{vk}_{\varnothing}$ as an output of Setup instead of as an output of Split.
- They have a separate algorithm Sign and $\mathsf{Sign}_{\mathsf{abo}}$ for the different types of signing keys.

Our notational changes allow us to state the security properties more concisely.

3.2 Construction

Using these building blocks, we now construct a fixed transcript garbling scheme $(\mathsf{Garble}, \mathsf{Eval})$ for RAM machines.

Let $M = (\Sigma, Q, Y, C)$ be a RAM machine. Recall that the transition functions C has two inputs – an internal state $q \in Q$ and a memory symbol $s \in \Sigma$ – and produces either an "official" output $y \in Y$, or a tuple $(q', \mathsf{op}) \in Q \times (\mathbb{N} \times \{\mathsf{Read}, \mathsf{Write}\} \times \Sigma)$.

Construction 3.1. *We define* Garble *and* Eval *such that:*

- $\mathsf{Garble}(M, x, T, S)$ *first samples* $\mathsf{Acc.PP}, \mathsf{ac}_0, \mathsf{store}_0 \leftarrow \mathsf{SetupAcc}(1^\lambda)$ *and* $\mathsf{Itr.PP}, \mathsf{itr}_0 \leftarrow \mathsf{Itr.Setup}(1^\lambda)$. *It also samples a puncturable PRF* F*, and splittable signature keys* $(\mathsf{sk}_0^A, \mathsf{vk}_0^A) \leftarrow \mathsf{SetupSpl}(1^\lambda; F(0))$*, and outputs* $(\tilde{M}, \tilde{x})$[2]*, defined as follows:*
  - $\tilde{M}$ *is just the circuit* $\mathsf{i}\mathcal{O}(\tilde{C})$*, where* $\tilde{C}$ *is a circuit defined in Algorithm 1.*
  - $\tilde{x}$ *is* $(q_0, \mathsf{Read}(0), \mathsf{ac}_x, \mathsf{itr}_0, \mathsf{Sign}(\mathsf{sk}_0^A, (q_0, \mathsf{Read}(0), \mathsf{ac}_x, \mathsf{itr}_0)), \mathsf{store}_x)$*, where* $\mathsf{op}_0 = \mathsf{Read}(0)$ *and* ac_x *and* store_x *are obtained as follows: Let* $a_1 < \cdots < a_{|x|}$ *be the set of addresses* a *for which* $x(a) \neq \epsilon$*. Now define* $\mathsf{ac}_0^x = \mathsf{ac}_0$*,* $\mathsf{store}_0^x = \mathsf{store}_0$*, and for* $i = 1, \ldots, |x|$*, define*
    1. $(\mathsf{store}_i^x, \mathsf{aux}_i^x) = \mathsf{Acc.Update}(\mathsf{Acc.PP}, \mathsf{store}_{i-1}^x, \mathsf{Write}(a_i \mapsto x(a_i)))$
    2. $(s_i, \mathsf{ac}_i^x) = \mathsf{Acc.LocalUpdate}(\mathsf{Acc.PP}, \mathsf{ac}_{i-1}^x, \mathsf{aux}_i^x)$

    We define $\mathsf{ac}_x = \mathsf{ac}_{|x|}^x$ *and similarly* $\mathsf{store}_x = \mathsf{store}_{|x|}^x$.
- *Given* $\tilde{M}$ *and* $\tilde{x} = (q_0, \mathsf{op}_0, \mathsf{ac}_x, \mathsf{itr}_0, \sigma_0), \mathsf{store}_x)$*, one computes* $\mathsf{Eval}(\tilde{M}, \tilde{x})$ *by repeating the following steps for* $i \in \{1, 2, \ldots\}$ *until termination.*
  1. $\mathsf{store}_i, \mathsf{aux}_{i-1} \leftarrow \mathsf{Acc.Update}(\mathsf{Acc.PP}, \mathsf{store}_{i-1}, \mathsf{op}_{i-1})$
  2. *Compute* $\mathsf{out}_i \leftarrow \tilde{M}(i-1, q_{i-1}, \mathsf{op}_{i-1}, \mathsf{ac}_{i-1}, \mathsf{itr}_{i-1}, \sigma_{i-1}, \mathsf{aux}_{i-1})$*. If* $\mathsf{out}_i \in Y \cup \{\perp\}$*, halt and output* out_i*.*
     Otherwise, parse out_i *as* $(q_i, \mathsf{op}_i, \mathsf{ac}_i, \mathsf{itr}_i, \sigma_i)$.

[2] To exactly match the syntax of a garbling scheme, the public parameters $\mathsf{Acc.PP}$ and $\mathsf{Itr.PP}$ may be placed either as part of the garbled machine or the garbled input.

Input: Timestamp t, state q, memory operation op, accumulator ac, iterator itr, signature σ, auxiliary information aux

Data: Transition function C_0, Puncturable PRF F, accumulator public parameters $\mathsf{Acc.PP}$, iterator public parameters $\mathsf{Itr.PP}$

if $t > T$ *or* $t < 0$ **then return** $\perp$;

$(\mathsf{vk}_t^A, \mathsf{sk}_t^A) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(t))$;

if $\mathsf{Spl.Verify}(\mathsf{vk}_t^A, (q, \mathsf{op}, \mathsf{ac}, \mathsf{itr}), \sigma) = 0$ **then return** $\perp$;

Parse $\mathsf{Acc.LocalUpdate}(\mathsf{Acc.PP}, \mathsf{ac}, \mathsf{op}, \mathsf{aux})$ as (s, ac') or else **return** $\perp$;

if $C_0(q, s) = y \in Y$ **then return** y;

Parse $C_0(q, s)$ as (q', op');

$(\mathsf{vk}_{t+1}^A, \mathsf{sk}_{t+1}^A) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(t+1))$;

$\mathsf{itr}' \leftarrow \mathsf{Itr.Iterate}(\mathsf{Itr.PP}, \mathsf{itr}, (q, \mathsf{op}, \mathsf{ac}, \mathsf{itr}, \mathsf{aux}))$;

$\sigma' \leftarrow \mathsf{Spl.Sign}(\mathsf{sk}_{t+1}^A, (q', \mathsf{op}', \mathsf{ac}', \mathsf{itr}'))$;

return $(q', \mathsf{op}', \mathsf{ac}', \mathsf{itr}', \sigma')$;

Algorithm 1: Circuit $\tilde{C}$

3.3 Proof of Security

Theorem 3.2. *If* Spl *is a splittable signature scheme,* Itr *is a cryptographic iterator,* Acc *is a positional accumulator, and* $\mathsf{i}\mathcal{O}$ *is an indistinguishability obfuscator, then Construction 3.1 is a fully succinct, efficient, fixed-transcript secure garbling scheme for RAM machines.*

Proof. Correctness, efficiency, and succinctness are easy to see. Correctness follows from the correctness of the splittable signatures, cryptographic iterators, and positional accumulators. Efficiency follows from the fact that the size of $\tilde{C}$ depends only polylogarithmically on the time bound T. Succinctness follows from the fact that store_x will have $\tilde{O}(|x|)$ non-empty addresses, and is represented succinctly.

We show a sequence of indistinguishable hybrid distributions starting with $\mathsf{Garble}(M_0, x)$ and ending with $\mathsf{Garble}(M_1, x)$. These hybrids are essentially identical to the ones in [KLW15], so we just give an overview of these hybrids.

Hybrids Overview At a high level, we only modify the circuit C in our hybrids, switching the execution from machine M_0 (with transition function C_0) to machine M_1 (with transition function C_1). Specifically, we use the variables in the definition of Eval to refer to the honest evaluation of $M_0(x)$ or $M_1(x)$. Since the transcripts are the same, we don't need to distinguish which execution we refer to.

1. We add a new branch to C. Instead of just checking that vk_t^A accepts $((q, \mathsf{op}, \mathsf{ac}, \mathsf{itr}), \sigma)$, we also check (when $t \leq T$) against the key vk_t^B, which is derived from a different puncturable PRF G. When only vk_t^B accepts $((q, \mathsf{op}, \mathsf{ac}, \mathsf{itr}), \sigma)$, we proceed as before except that we compute with C_1 instead of C_0, and we sign outputs with sk_{t+1}^B instead of with sk_{t+1}^A.

   The indistinguishability of this change follows by $O(t)$ applications of the indistinguishability of punctured keys, together with the security of $\mathsf{i}\mathcal{O}$.

2. We hard-code vk_0^A and vk_0^B, and puncture F and G at $\{0\}$. This change preserves functionality and is hence indistinguishable by $\mathsf{i}\mathcal{O}$.

3. We replace vk_0^A and vk_0^B by keys punctured on the sets $\mathcal{M} \setminus \{(q_0, \mathsf{op}_0, \mathsf{ac}_0, \mathsf{itr}_0)\}$ and $\{(q_0, \mathsf{op}_0, \mathsf{ac}_0, \mathsf{itr}_0)\}$ respectively. These changes are indistinguishable by the indistinguishability of punctured keys.

4. We generate $\mathsf{Acc.PP}$ using $\mathsf{SetupAccEnforceUpdate}$ so that aux_0 is the only value for aux such that $\mathsf{LocalUpdate}(\mathsf{Acc.PP}, \mathsf{ac}_0, \mathsf{op}_0, \mathsf{aux}) \neq \perp$, and is in fact equal to s_0, ac_1. This is indistinguishable by the positional accumulator's setup indistinguishability.

5. We are guaranteed that $C_0(q_0, s_0) = C_1(q_0, s_1)$, so we modify C so that it uses C_1 in both the 'A' *and* the 'B' branch at time 0, which preserves functionality and is thus indistinguishable by $i\mathcal{O}$.

6. We generate $\mathsf{Acc.PP}$ normally, which is an indistinguishable change due to the positional accumulator's setup indistinguishability.

7. We modify C so that at time 0, instead of signing with sk_1^A in branch 'A' and sk_1^B in branch B, we do the same thing in both branches. Namely, we use sk_1^A if and only if $(q, \mathsf{op}, \mathsf{ac}, \mathsf{itr}) = (q_0, \mathsf{op}_0, \mathsf{ac}_0, \mathsf{itr}_0)$. This is functionally equivalent because vk_0^A and vk_0^B accept disjoint sets of messages, and hence this change is indistinguishable by $i\mathcal{O}$. Note the 'A' branch and 'B' branch are now identical.

8. We generate $\mathsf{Itr.PP}$ using $\mathsf{SetupEnforce}$ so that $\mathsf{itr}' = \mathsf{itr}_1$ if and only if $(q, \mathsf{op}, \mathsf{ac}, \mathsf{itr}, \mathsf{aux})$ is equal to $(q_0, \mathsf{op}_0, \mathsf{ac}_0, \mathsf{itr}_0, \mathsf{aux}_0)$. This change is indistinguishable by the iterator's setup indistinguishability.

9. Instead of choosing whether to use sk_1^A or sk_1^B based on the value of $(q, \mathsf{op}, \mathsf{ac}, \mathsf{itr})$, we choose based on the value of $(q', \mathsf{op}', \mathsf{ac}', \mathsf{itr}')$. This is functionally equivalent because itr' is equal to itr_1 (and in fact $(q', \mathsf{op}', \mathsf{ac}')$ is equal to $(q_1, \mathsf{op}_1, \mathsf{ac}_1)$) if and only if $(q, \mathsf{op}, \mathsf{ac}, \mathsf{itr}, \mathsf{aux}) = (q_0, \mathsf{op}_0, \mathsf{ac}_0, \mathsf{itr}_0, \mathsf{aux}_0)$, and therefore this change is indistinguishable by the security of $i\mathcal{O}$.

10. We generate $\mathsf{Itr.PP}$ normally, which is indistinguishable by the iterator's indistinguishability of setup.

11. Instead of checking whether the signature σ on $(q, \mathsf{ac}, \mathsf{itr})$ verifies under one of vk_0^A (which is punctured at $\mathcal{M} \setminus \{(q_0, \mathsf{op}_0, \mathsf{ac}_0, \mathsf{itr}_0)\}$) and vk_0^B (which is punctured at $\{(q_0, \mathsf{op}_0, \mathsf{ac}_0, \mathsf{itr}_0)\}$), we only check that it verifies under the *unpunctured* vk_0^A. This is indistinguishable by the splittable signature's splitting indistinguishability property.

12. We unpuncture F and G at 0 and un-hardcode vk_0^A and vk_0^B. This is functionally equivalent and hence indistinguishable by $i\mathcal{O}$.

13. We repeat steps 2 through 12 for timestamps 1 through the worst-case running time bound T instead of just for timestamp 0 as was described above. In this way, we progressively change the computation from using C_0 (M_0's transition function) to C_1 (M_1's transition function), starting at the beginning of the computation.

□

4 Fixed Memory Garbling

We now use a fixed transcript garbling scheme to satisfy a slightly stronger notion which we call fixed-memory garbling. In fixed-memory garbling, the garblings of two different machines are indistinguishable as long as the memory accesses are the same. Notably, it is possible for the two machines to have differing local states.

Definition 4.1 (Fixed Memory Security)**.** A garbling scheme $(\mathsf{Garble}, \mathsf{Eval})$ is said to be *fixed-memory secure* if

$$\mathsf{Garble}(M_0, x, T_0, S) \approx \mathsf{Garble}(M_1, x, T_1, S)$$

for RAM machines $M_0 = (\Sigma, Q, Y, C_0)$ and $M_1 = (\Sigma, Q, Y, C_1)$ and a memory configuration $x : \mathbb{N} \to \Sigma$ whenever the following conditions hold:

- $M_0(x) = M_1(x)$
- $|C_0| = |C_1|$
- The sequence of memory operations made by M_0 on x is identical to that of M_1 on x. In particular they also have the same time and space complexities, and at each time M_0 and M_1 have the same memory configurations.

4.1 Construction

Given a garbling scheme $(\mathsf{Garble}', \mathsf{Eval}')$ satisfying fixed transcript security, we build a garbling scheme $(\mathsf{Garble}, \mathsf{Eval})$ satisfying fixed-memory security. All we need to do is mask the internal state for each timestamp with a different pseudorandom value.

Construction 4.1. *We define* $(\mathsf{Garble}, \mathsf{Eval})$ *such that:*

- $\mathsf{Garble}(M, x, T, S)$ *samples a puncturable PRF* F*, and outputs* $\mathsf{Garble}'(M', x, T, S)$*, where* M' *is a RAM program whose transition function is given by* C'*, defined in Algorithm 2.*
- Eval *is the same as* Eval'*.*

Input: state p, memory symbol s
Data: Puncturable PRF F, underlying transition function C
if $p = \perp$ **then** $t \leftarrow 0, q \leftarrow \perp$;
else
  Parse p as (t, c_q);
  $q \leftarrow F(t) \oplus c_q$;
end
if $C(q, s) = y \in Y$ **then return** y;
else
  Parse $C(q, s)$ as (q', op);
  return $((t+1, F(t+1) \oplus q'), \mathsf{op})$;
end

Algorithm 2: Transition function C'

4.2 Proof of Security

Theorem 4.2. *If* $(\mathsf{Garble}', \mathsf{Eval}')$ *is a fixed transcript secure garbling scheme, then Construction 4.1 defines a fully succinct, efficient, fixed memory secure garbling scheme for RAM machines.*

Proof. Let M_0, M_1 be two RAM machines that satisfy the preconditions of fixed-memory security. Let C_0, C_1 be the transition functions of the two machines, respectively. We show a sequence of $t_x + 1$ hybrid distributions $H_0 \approx \cdots \approx H_{t_x}$, where t_x is the running time of M_0 on x (which is the same as the running time of M_1 on x).

Hybrid H_i is defined as $\mathsf{Garble}'(M_{0,i}, x)$, where $M_{0,i}$ is a RAM machine with transition function $C_{0,i}$, defined in Algorithm 3. $M_{0,i}$ runs M_0 for the first $t_x - i$ steps, and then runs M_1 for the last i steps, starting with the hard-coded internal state $q^* = q_{1,t_x-i}$. Here q_{1,t_x-i} is defined as the $t_x - i^{th}$ internal state if M_1 were to be run on x.

Evidently H_0 is indistinguishable from $\mathsf{Garble}(M_0, x)$ and H_{t_x} is indistinguishable from $\mathsf{Garble}(M_1, x)$ by the fixed transcript security of Garble'. In order to complete the proof that $\mathsf{Garble}(M_0, x) \approx \mathsf{Garble}(M_1, x)$, we just need to show the following claim.

Claim 4.2.1. *For all* i *such that* $0 \leq i < t_x$*,* $H_i \approx H_{i+1}$*.*

Proof. We give hybrid distributions $H_{i,1}$ and $H_{i,2}$ such that $H_i \approx H_{i,1} \approx H_{i,2} \approx H_{i+1}$.

Hybrid $H_{i,1}$ is defined as $\mathsf{Garble}'(M_{0,i,1}, x)$ where $M_{0,i,1}$'s transition function is $C_{0,i,1}$, given in Algorithm 4. $C_{0,i,1}$ never evaluates F' at $t_x - i$. Instead, it has the hard-coded constant $c^* = F(t_x - i) \oplus q_{0,t_x-i}$. Here q_{0,t_x-i} is defined as the $t_x - i^{th}$ internal state when M_0 is executed on x.

Hybrid $H_{i,2}$ differs from hybrid $H_{i,1}$ only in that c^* is hard-coded as $F(t_x - i) \oplus q_{1,t_x-i}$ instead of as $F(t_x - i) \oplus q_{0,t_x-i}$. Here q_{1,t_x-i} is defined as the $t_x - i^{th}$ internal state when M_1 is executed on x.

Input: state p, memory symbol s
Data: Puncturable PRF F, underlying transition functions C_0 and C_1, string q^* representing an internal state of M_1

if $p = \bot$ **then** $t \leftarrow 0, q \leftarrow \bot$;
else
  Parse p as (t, c_q);
  if $t = t_x - i$ **then** $q \leftarrow q^*$;
  else $q \leftarrow F'(t) \oplus c_q$;
end
if $t < t_x - i$ **then** $C := C_0$;
else $C := C_1$;
if $C(q, s) = y \in Y$ **then return** y;
else
  Parse $C(q, s)$ as (q', op);
  return $((t + 1, F(t + 1) \oplus q'), \mathsf{op})$;
end

Algorithm 3: Transition function $C_{0,i}$

We now need to show that

$$H_i \approx H_{i,1} \approx H_{i,2} \approx H_{i+1}$$

The first and third transitions are shown via reduction to the fixed transcript security of Garble'. The second transition is shown via reduction to the pseudorandomness of F at the selectively punctured point $t_x - i$. □

This concludes the proof of Theorem 4.2. □

5 Fixed Address Garbling

We now use a fixed memory garbling scheme to construct a slightly stronger notion of garbling. Namely, we will now hide the *data* in memory, but not yet the addresses which are accessed. As discussed in the Introduction, in applications where the memory access pattern is known not to leak sensitive information, this notion of garbling may be significantly more efficient. In particular, it preserves the efficacy of cache, for which real-world RAM programs are extensively optimized.

Definition 5.1. A garbling scheme $(\mathsf{Garble}, \mathsf{Eval})$ is said to be fixed-address secure if $\mathsf{Garble}(M_0, x_0, T_0, S) \approx \mathsf{Garble}(M_1, x_1, T_1, S)$ whenever the following conditions hold:

- $M_0(x_0) = M_1(x_1)$
- $|M_0| = |M_1|$
- The space usages of $M_0(x)$ and $M_1(x)$ are both less than S.
- The addresses of x_0 containing non-ϵ symbols are the same as those of x_1.
- The sequence of addresses accessed by M_0 on x_0 is identical to that of M_1 on x_1.

5.1 Construction

Given a garbling scheme $(\mathsf{Garble}', \mathsf{Eval}')$ satisfying fixed-memory security, we build a garbling scheme $(\mathsf{Garble}, \mathsf{Eval})$ satisfying fixed-address security.

Input: state p, memory symbol s
Data: Punctured PRF $F' = F\{t_x - i\}$, underlying transition function C_0, plaintext $q^* = q_{1,t_x-i}$ and ciphertext c^*

if $p = \bot$ **then** $t \leftarrow 0, q \leftarrow \bot$;
else
  Parse p as (t, c_q);
  if $t = t_x - i$ **then** $q \leftarrow q^*$;
  else $q \leftarrow F'(t) \oplus c_q$;
end
if $t < t_x - i$ **then** $C := C_0$;
else $C := C_1$;
if $C(q, s) = y \in Y$ **then return** y;
Parse $C(q, s)$ as (q', op);
if $t = t_x - i - 1$ **then return** $((t + 1, c^*), \mathsf{op})$;
else return $((t + 1, F(t + 1) \oplus q'), \mathsf{op})$;

Algorithm 4: Transition function $C_{0,i,1}$

Overview. Our construction of $\mathsf{Garble}(M, x, T, S)$ applies Garble' to a transformed version of the machine M and a correspondingly transformed of the input x. The transformed machine, which we will denote by M', differs from M in three ways:

- M' executes two copies of M in parallel (thereby using twice as much memory). We think of these as an 'A' execution and a 'B' execution. We think of the external storage of M' as correspondingly consisting of an 'A' track and a 'B' track. We implement the 'A' track as the set of all *even* addresses and the 'B' track as the set of all *odd* addresses.
- M' writes a timestamp alongside each value it writes, indicating the time at which the value is written.
- M' authenticates each value it writes: instead of writing (t, v) to an address a, it writes $(t, F(t\|a) \oplus v)$, where F is a puncturable pseudorandom function.

The initial memory configuration $x : \mathbb{N} \to \Sigma$ must be transformed in an analogous way to obtain x'. Essentially, x' stores a copy of x on both track 'A' and on track 'B', appropriately authenticated. That is, the ith bit of x is stored at locations $2i$ and $2i + 1$, authenticated with timestamp $t = 0$.

Construction 5.1. *We define* $(\mathsf{Garble}, \mathsf{Eval})$ *such that:*

- $\mathsf{Garble}(M, x, T, S)$ *samples a puncturable pseudorandom function* F *mapping* $\{0,1\}^\lambda \to \{0,1\}$*, and outputs* $\mathsf{Garble}'(M', x', 2T, 2S)$*, where* M' *and* x' *are defined below.*
- Eval *is the same as* Eval'*.*

Definition of M' The RAM machine M' is described in Algorithm 6, and makes use of a helper function Access, defined in Algorithm 5. Both M' and Access are presented in reactive form, describing the actions taken per activation. This is done to demonstrate more immediate correspondence to the syntax of RAM machines in Section 2.1. Still, it is helpful to keep the above informal description in mind.

Definition of x' The initial memory x' is defined as

$$x' : \mathbb{N} \to \Sigma$$

$$x'(a) = \begin{cases} (0,\ x(\lfloor a/2 \rfloor)\ \oplus\ F(0\|a)) & \text{if } x(\lfloor a/2 \rfloor) \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

Data: Puncturable PRF F

On input address a, value s, operation op, time t, track k **from the RAM machine do:**

  Let a' be $2a$ if $k =$'A', and $2a+1$ if $k =$'B';

  If $\mathsf{op} = \mathsf{Read}$ **then** Output (Read, a') to the memory;

  If $\mathsf{op} = \mathsf{Write}$ **then** Output (Write, a', c) to the memory, where $c \leftarrow F(t\|a') \oplus s$;

On input s **from the memory do:**

  parse s as (t', c);

  let a' be the address given by the RAM machine in the previous activation;

  Output $F(t'\|a') \oplus c$ to the RAM machine;

Algorithm 5: Access

Data: RAM machine transition function C, initial state q_0

In first activation:

  $(q_A, s_A, a_A, \mathsf{op}_A), (q_B, s_B, a_B, \mathsf{op}_B) \leftarrow ((q_0, \perp, 0, \mathsf{Read}), (q_0, \perp, 0, \mathsf{Read}))$;

  $t \leftarrow 0$, $k \leftarrow$'A';

  activate $\mathsf{Access}(a_k, s_k, \mathsf{op}_k, t, k)$;

On input s **from** Access **do:**

  $s_k \leftarrow s$;

  $\mathsf{out} \leftarrow C(q_k, s_k)$;

  If $k =$'A' and $\mathsf{out} \in Y$ **then** output out and halt;

  parse $q_k, a_k, s_k, \mathsf{op}_k \leftarrow \mathsf{out}$;

  If $k =$'A' **then** $k \leftarrow$'B'; **else** $k \leftarrow$'A' and $t \leftarrow t+1$;

  activate $\mathsf{Access}(a_k, s_k, \mathsf{op}_k, t, k)$;

Algorithm 6: M'

5.2 Proof of Security

Theorem 5.2. *If* $(\mathsf{Garble}', \mathsf{Eval}')$ *is a fully succinct, fixed memory secure garbling scheme, and if one-way functions (and hence puncturable PRFs) exist, then Construction 5.1 defines a fully succinct, fixed address secure garbling scheme for RAM machines.*

Proof. Let M_0, x_0, T_0, S and M_1, x_1, T_1, S be RAM machines, initial inputs, runtime bounds and space bound as in the premise of the definition of fixed-address security. Let $M_{00} = M'_0$ and $x_{00} = x'_0$ denote the transformed versions of M_0 and x_0, and let M_{11}, x_{11} denote the transformed versions of M_1 and x_1. We define hybrid RAM machine M_{01} and input x_{01}, and let $H_{01} = \mathsf{Garble}'(M_{01}, x_{01}, T'_0, S')$. We then show that

$$\mathsf{Garble}(M_0, x_0, T_0, S) \approx H_{01} \approx \mathsf{Garble}(M_1, x_1, T_0, S).$$

Definition of M_{01}. Suppose M_0 and M_1 have transition functions C_0 and C_1 respectively. Informally, M_{01} just executes M_0 on track 'A' and executes M_1 on track 'B'. Formally, M_{01} is defined in Algorithm 7. Here q_{00}, q_{01} are the initial states of M_0, M_1, respectively.

Definition of x_{01}. Informally, x_{01} has x_0 on track 'A' and x_1 on track 'B'. Formally:

$$x_{01}(a) = \begin{cases} (0,\ x_0(\frac{a}{2}) \ \oplus\ F(0\|a)) & \text{if } a \text{ is even and } x_0(\frac{a}{2}) \neq \epsilon \\ (0,\ x_1(\frac{a-1}{2}) \ \oplus\ F(0\|a)) & \text{if } a \text{ is odd and } x_1(\frac{a-1}{2}) \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

Lemma 5.3. $\mathsf{Garble}(M_0, x_0) \equiv \mathsf{Garble}'(M_{00}, x_{00}) \approx \mathsf{Garble}'(M_{01}, x_{01})$

Data: RAM transition functions C_0 and C_1 of machines M_0 and M_1 respectively, and their initial states q_{00} and q_{01}

In first activation:

  $(q_A, s_A, a_A, \mathsf{op}_A), (q_B, s_B, a_B, \mathsf{op}_B) \leftarrow ((q_{00}, \bot, 0, \mathsf{Read}), (q_{01}, \bot, 0, \mathsf{Read}))$;

  $t \leftarrow 0$, $k \leftarrow$'A';

  activate $\mathsf{Access}(a_k, s_k, \mathsf{op}_k, t, k)$;

On input s **from** Access **do:**

  $s_k \leftarrow s$;

  If $k =$'A' **then** $\mathsf{out} \leftarrow C_0(q_k, s_k)$; **else** $\mathsf{out} \leftarrow C_1(q_k, s_k)$;

  If $k =$'A' and $\mathsf{out} \in Y$ **then** output out and halt;

  parse $q_k, a_k, s_k, \mathsf{op}_k \leftarrow \mathsf{out}$;

  If $k =$'A' **then** $k \leftarrow$'B'; **else** $k \leftarrow$'A' and $t \leftarrow t + 1$;

  activate $\mathsf{Access}(a_k, s_k, \mathsf{op}_k, t, k)$;

Algorithm 7: RAM machine M_{01}

Proof. We show a sequence of $t^* + 1$ indistinguishable hybrid distributions $H_{00,i} = \mathsf{Garble}'(M_{00,i}, x_{01})$ (where $M_{00,i}$ is a RAM machine to be defined) for $i = 0, \ldots, t^*$ such that

$$\mathsf{Garble}'(M_{00}, x_{00}) \approx H_{00,0} \approx \cdots \approx H_{00,t^*} \approx \mathsf{Garble}'(M_{01}, x_{01}).$$

Definition of $M_{00,i}$ Informally, $M_{00,i}$ executes M_0 on track A and M_1 on track B, but only for the first i steps of computation. After this, $M_{00,i}$ ignores the contents of track B. Instead, $M_{00,i}$ only executes M_0 on track A, but writes the same underlying symbols (masked independently) to both track A and track B. Formally $M_{00,i}$ is defined in Algorithm 8. (The only difference between $M_{00,i}$ and M_{01} is in line 11.)

Data: RAM transition functions C_0 and C_1, initial states q_{00}, q_{01}

In first activation:

  $(q_A, s_A, a_A, \mathsf{op}_A), (q_B, s_B, a_B, \mathsf{op}_B) \leftarrow ((q_{00}, \bot, 0, \mathsf{Read}), (q_{01}, \bot, 0, \mathsf{Read}))$;

  $t \leftarrow 0$, $k \leftarrow$'A';

  activate $\mathsf{Access}(a_k, s_k, \mathsf{op}_k, t, k)$;

On input s **from** Access **do:**

  $s_k \leftarrow s$;

  If $k =$'A' **then** $\mathsf{out} \leftarrow C_0(q_k, s_k)$; **else** $\mathsf{out} \leftarrow C_1(q_k, s_k)$;

  If $k =$'A' and $\mathsf{out} \in Y$ **then** output out and halt;

  parse $q_k, a_k, s_k, \mathsf{op}_k \leftarrow \mathsf{out}$;

  If $k =$'A' **then** $k \leftarrow$'B'; **else** $k \leftarrow$'A' and $t \leftarrow t + 1$;

  If $k =$'A' $\vee$ ($k =$'B' $\wedge$ $t \leq i$) **then** activate $\mathsf{Access}(a_k, s_k, \mathsf{op}_k, t, k)$; **else** $\mathsf{Access}(a_A, s_A, \mathsf{op}_A, t, B)$;

Algorithm 8: Algorithm $M_{00,i}$

Claim 5.3.1. $\mathsf{Garble}'(M_{00}, x_{00}) \approx \mathsf{Garble}'(M_{00,0}, x_{01})$

Proof. First, $\mathsf{Garble}'(M_{00}, x_{00}) \approx \mathsf{Garble}'(M_{00,0}, x_{00})$ by the fixed memory security of Garble'. Indeed, both of these perform the same memory operations – in both M_{00} and $M_{00,0}$ the memory operations are determined solely by evaluating M_0 on track A.

We next need to show that $\mathsf{Garble}'(M_{00,0}, x_{00}) \approx \mathsf{Garble}'(M_{00}, x_{01})$. This again proceeds by several hybrids. We illustrate how to indistinguishably change the underlying symbol at address a_i of track B (which is physically stored at address $2a_i + 1$ of x_{00}) from $x_0(a_i)$ to $x_1(a_i)$. In other words, we need to change $x_{00}(2a_i + 1)$ from $(0, F(0\|2a_i + 1) \oplus x_0(a_i))$ to $(0, F(0\|2a_i + 1) \oplus x_1(a_i))$.

First, we indistinguishably puncture F at $\{0\|2a_i+1\}$ using the observation that $M_{00,0}$ never actually needs to decrypt *any* value in track B. Thus, we can easily puncture F at $\{0\|2a_i+1\}$ in $M_{00,0}$ without changing the memory operations. Indistinguishable follows from the fixed memory security of Garble'. Using a now standard technique, we now apply the pseudorandomness of F at $(0\|2a_i+1)$ to change $x_{00}(2a_i+1)$ to $(0, F(0\|2a_i+1) \oplus x_1(a_i))$. Afterwards we unpuncture F, which is indistinguishable by the same fixed memory security argument as when we punctured F.

We repeat these steps for each a_i such that $x_0(a_i) \neq x_1(a_i)$. □

Claim 5.3.2. *For each* $1 \leq i \leq t^*$, $\mathsf{Garble}'(M_{00,i-1}, x_{01}) \approx \mathsf{Garble}'(M_{00,i}, x_{01})$

Proof. We make a sequence of changes to $M_{00,i-1}$ such that the induced changes to $\mathsf{Garble}'(M, x_{01})$ are indistinguishable and eventually we have changed $M_{00,i-1}$ into $M_{00,i}$.

Let s_i^0 denote the value written by M_0 at time i, and let s_i^1 denote the value written by M_1 at time i. Let a_i denote the corresponding address to which M_0 and M_1 write at time i. This address is well-defined because M_0 accesses the same addresses on input x_0 as M_1 does on input x_1.

We first modify $M_{00,i-1}$ so that at time i, it writes (i, c_i^0) to address a_i on track B, where c_i^0 is a hard-coded ciphertext equal to $F(i\|2a_i+1) \oplus s_i^0$. We also change it to only use a punctured $F' = F\{i\|2a_i+1\}$, as in Algorithm 9. This is indistinguishable by the fixed memory security of Garble'. Indeed, it changes neither the addresses accessed nor the values written to memory.

We next change the hard-coded value of c_i^0 from $(i, F(i\|2a_i+1) \oplus s_i^0)$ to $(i, F(i\|2a_i+1) \oplus s_i^1)$. The indistinguishability of this change follows from the pseudorandomness of F at the selectively punctured point $i\|2a_i+1$.

After these changes, $M_{00,i-1}$ accesses the same addresses and writes the same values as $M_{00,i}$ on x_{01}. So by the fixed memory security of Garble', $\mathsf{Garble}'(M, x_{01})$ is indistinguishable from $\mathsf{Garble}'(M_{00,i}, x_{01})$.

Data: RAM transition functions C_0 and C_1, initial states q_{00}, q_{01}, ciphertext c^*, address a^*
1 **In first activation:**
2 $(q_A, s_A, a_A, \mathsf{op}_A), (q_B, s_B, a_B, \mathsf{op}_B) \leftarrow ((q_{00}, \bot, 0, \mathsf{Read}), (q_{01}, \bot, 0, \mathsf{Read}))$;
3 $t \leftarrow 0$, $k \leftarrow$'A';
4 activate $\mathsf{Access}(a_k, s_k, \mathsf{op}_k, t, k)$;
5 **On input** s **from** Access **do:**
6 $s_k \leftarrow s$;
7 **If** $k =$'A' **then** $\mathsf{out} \leftarrow C_0(q_k, s_k)$; **else** $\mathsf{out} \leftarrow C_1(q_k, s_k)$;
8 **If** $k =$'A' and $\mathsf{out} \in Y$ **then** output out and halt;
9 parse $q_k, a_k, s_k, \mathsf{op}_k \leftarrow \mathsf{out}$;
10 **If** $k =$'A' **then** $k \leftarrow$'B'; **else** $k \leftarrow$'A' and $t \leftarrow t+1$;
11 **If** $k =$'A' $\vee$ ($k =$'B' $\wedge$ $t < i$) **then** activate $\mathsf{Access}(a_k, s_k, \mathsf{op}_k, t, k)$;
12 **else if** $t = i$ **then** output $(\mathsf{Write}, s^*, a^*)$ directly to memory;
13 **else** activate $\mathsf{Access}(a_A, s_A, \mathsf{op}_A, t, B)$;

Algorithm 9: Intermediate machine M

□

Claim 5.3.3. $\mathsf{Garble}'(M_{00,t^*}, x_{01}) \approx \mathsf{Garble}'(M_{01}, x_{01})$

Proof. Lines 8 through 10 of M_{00,t^*} (described in Algorithm 8) are never activated on input x_{01} because M_0 terminates after t^* steps. It's easy to see then that M_{00,t^*} on input x_{01} accesses the same addresses and writes the same values as M_{01} on input x_{01}. So the claim follows from the fixed-memory security of Garble'. □

Lemma 5.3 follows by combining claims 5.3.1, 5.3.2, and 5.3.3. Recall that we wanted to show $\mathsf{Garble}'(M_{00}, x_{00}) \approx \mathsf{Garble}'(M_{01}, x_{01})$. Claim 5.3.1 showed that $\mathsf{Garble}'(M_{00}, x_{00}) \approx \mathsf{Garble}'(M_{00,0}, x_{01})$. Claim 5.3.2 showed that

$\mathsf{Garble}'(M_{00,0}, x_{01}) \approx \cdots \approx \mathsf{Garble}'(M_{00,t^*}, x_{01})$. Finally, Claim 5.3.3 showed that $\mathsf{Garble}'(M_{00,t^*}, x_{01}) \approx \mathsf{Garble}'(M_{01}, x_{01})$. □

Lemma 5.4. $\mathsf{Garble}'(M_{01}, x_{01}) \approx \mathsf{Garble}'(M_{11}, x_{11}) \equiv \mathsf{Garble}(M_1, x_1)$

Proof. This follows analogously and symmetrically to Lemma 5.3. □

The fixed-access security of Garble follows from Lemmas 5.3 and 5.4. □

6 Full Security

This section constructs a fully succinct and secure garbling scheme as in Definition 2.5. This is done by combining any fixed access garbling scheme with an oblivious RAM (ORAM) scheme that has a special property, called localized randomness (see informal presentation and motivation in the Introduction). We start by formally defining oblivious RAM schemes and localized randomness, and then present and prove security of the garbling scheme.

6.1 Oblivious RAM

Following Goldreich and Ostrovsky [GO96], we define an ORAM as a procedure OAccess, which is a randomized, stateful replacement for the memory accesses of a RAM machine. In other words, OAccess serves as a "layer" between a RAM machine and the external memory, interacting both with external memory and with the underlying RAM machine.

That is, an execution of and ORAM scheme is a sequence of activations, where an activation can be prompted by two types of input: One type is an input coming from the external memory. This input contains the value of an external memory cell. As a result, the ORAM scheme can either: (a) Generate an output value to the underlying RAM machine; this value will again be the contents of a memory cell. (b) Generate output value to the external memory without activating the underlying RAM machine at all. This value will be a value to be written to the currently read address, plus a new memory address to be read from.

The other type of input is a *memory access tuple* coming from the underlying RAM machine. Again, as a reaponse, the ORAM scheme can either generate an output value to the underlying RAM machine without returning any value to the external memory, or output a memory access tuple to the external memory.

Formally, an ORAM scheme is described by a function OAccess which maps $Q \times (O_\Sigma \cup \Sigma') \times \{0,1\}^{\ell_r} \to Q \times (\Sigma \cup O_{\Sigma'})$. Here Q is the set of states of the ORAM scheme and O_Σ is the set of memory operations with alphabet Σ. That is, OAccess takes for input a state q and randomness r, along with either a memory operation op coming from the RAM program or a value m coming from the external memory. It then generates a new state q' together with either a value s' to the RAM program, or a memory operation op′ to the external memory.

Given a RAM machine M and an ORAM scheme OAccess, the combined RAM machine M_{OAccess} is defined in the natural way. That is, the state space of M_{OAccess} is the cartesian product of the state spaces M and of OAccess, and each activation of M_{OAccess} starts with an activation of OAccess and potentially continues to an activation of M, as described above.

6.1.1 Localized-Randomness ORAM

We formally define the notion of localized-randomness ORAMs, introduced and motivated in the Introduction. Appendix A describes the path ORAM scheme of Chung and Pass [CP13] and argues that it has localized randomness.

Let OAccess be an ORAM scheme. For a RAM machine M and initial memory configuration s_0, consider an execution of the combined machine M_{OAccess} from initial memory configuration s_0. Let $\vec{r} = r_0, ..., r_t$ be

the random strings that $\mathsf{OAccess}$ takes as input in this execution, and let $\mathbf{a}_i$ denote the random variable describing the sequence of memory addresses accessed by M_{OAccess} at the ith activation.

We say that $\mathbf{a}_i$ depends on bit location j if there exist two different values of $\vec{r}$, differing only on the j^{th} bit, which cause $\mathbf{a}_i$ to have differing values in an execution of M_{OAccess}. Let S_i denote the set of bit locations that a_i depends on. A localized-randomness ORAM satisfies the following two properties.

Property 6.1. *For all RAM machines M, all initial memory configurations s_0, and all i, the corresponding set S_i is at most* $\mathrm{poly}(\lambda)$ *in size. Furthermore, S_i is efficiently computable as a function of (M, s_0, i). In addition, for $i \neq j$, S_i and S_j are disjoint.*

Property 6.2. *There is an efficient algorithm* $\mathsf{OSample}$ *such that* $\mathsf{OSample}(i)$ *and $\mathbf{a}_i$ are indistinguishable. Here, the running time of* $\mathsf{OSample}$ *must be polynomial in* $\log i$.

Note that, in a real execution of M_{OAccess}, the sequence $\mathbf{a}_i$ may be determined adaptively depending on the actual values contained in the memory locations accessed. Still, $\mathsf{OSample}$ should generate a sequence that is distributed indistinguishably, without knowing any of the values located in the accessed memory locations.

6.2 Construction

Our garbling scheme is very simple; essentially, we just layer the fixed address garbler on top of an ORAM scheme with localized randomness. A bit more specifically, given an ORAM scheme $\mathsf{OAccess}$ we proceed as so:

1. We sample a function F from a puncturable PRF family, and modify $\mathsf{OAccess}$ so that it generates its local randomness for step i by applying F to i and other data (see below for details). Let $\mathsf{OAccess}_F$ denote the resulting scheme.
2. We let the garbled machine be the result of applying the fixed-address garbler to the machine M_{OAccess_F}, where M is the 'plaintext' RAM machine.
3. We let the garbled input be the result of applying the fixed-address garbler to an initial memory configuration x' which is the valid encoding of the plaintext input x according to $\mathsf{OAccess}_F$.

We move to the full description of the scheme.

Construction 6.3. *Let* $(\mathsf{Garble}', \mathsf{Eval}')$ *be a fixed-address garbling scheme, and let* $\mathsf{OAccess}$ *be an ORAM scheme with localized randomness. We define* Garble *and* Eval *as so:*

- $\mathsf{Garble}(M, x, T, S)$ *samples a puncturable pseudorandom function F mapping $\{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}$ and outputs* $\mathsf{Garble}'(M', x', T', S')$*, where M' and x' are defined below. T' is $\tilde{O}(T)$ and S' is $\tilde{O}(S)$, and their exact values are determined by the time overhead and memory overhead of the ORAM that we use.*
- Eval *is identical to* Eval'.

Definition of M'. Because M' is an input to a fixed-access garbler, its precise formulation as a transition function doesn't matter. We instead describe M' in Algorithm 10 with procedural pseudocode using $\mathsf{OAccess}$ as a subroutine which transparently accesses many addresses of external memory and returns a memory symbol s. An alternate description, more closely matching a RAM machine's formal syntax, is given in Algorithm 18 on page 29, in Appendix B.

Let ℓ_q be the bit length of the ORAM local state, and let ℓ_r be the amount of randomness used by $\mathsf{OAccess}$. M' first reads the initial state of the ORAM scheme from a predetermined area of memory, namely the first ℓ_q bits of memory.

Data: Puncturable PRF F, RAM machine M with transition function C

Read an initial ORAM state q_{ORAM} from addresses $0, \ldots, \ell_q - 1$;
$\mathsf{out} := (q_0, \mathsf{Read}(0))$;
$i := 0$;
while $\mathsf{out} \notin Y$ **do**
    Parse out as (q, op);
    Run $(q_{ORAM}, s) \leftarrow \mathsf{OAccess}(q_{ORAM}, \mathsf{op}; F(i,0), \ldots, F(i,\ell_r))$, but modify the generated external memory operations by adding ℓ_q to the accessed addresses;
    `// The above denotes activating OAccess repeatedly until it returns a memory value` s`, and not an external memory operation.`
    $\mathsf{out} \leftarrow C(q, s)$;
    $i \leftarrow i + 1$;
end
return out;

Algorithm 10: Machine M', the garbled version of M. See alternative presentation in Algorithm 18.

Definition of x'. Informally, we initialize our ORAM and add the contents of x, which results in a memory configuration X and an ORAM state $q_{ORAM,x}$. We then define x' as the concatenation $q_{ORAM,x} \| X$.

6.3 Security Proof

Theorem 6.4. *If $(\mathsf{Garble}', \mathsf{Eval}')$ is a fully succinct, fixed address secure garbler, and F, G are drawn from puncturable PRF families, then Construction 6.3 defines a fully succinct, fully secure garbling scheme for RAM machines.*

Proof Overview We proceed through a sequence of hybrids, changing M' so that instead of running $\mathsf{OAccess}$ it runs the dummy address generator $\mathsf{OSample}$. We make this change one timestep at a time; starting with the case when the timestamp i is t_x (the running time of M on x), and then working our way down to the case when $i = 0$.

To prove these hybrids indistinguishable, we make crucial use of the ORAM's localized randomness. Indeed, locality allows us to isolate the randomness that determines the particular addresses we are trying to change. In conjunction with our fixed address garbler, which lets us ignore low-level RAM machine details, we then use the punctured programming method to change the addresses accessed. Finally, as an edge case of this same technique, we change x' to be simulatable given $|x|$, the number of non-empty addresses of x.

Proof. For any RAM machine M and input x with running time t_x and space S, we give a sequence of $t_x + 2$ indistinguishable hybrid distributions H_0 through H_{t_x+1}. For $0 \le i \le t_x$, hybrid H_i is defined as $\mathsf{Garble}'(M_i, x')$. Hybrid H_{t_x+1} is defined as $\mathsf{Garble}'(M_{t_x+1}, x_{t_x+1})$, where M_{t_x+1} and x_{t_x+1} are to be defined, and will depend only on $M(x)$, $|x|$, and t_x.

H_i: In hybrid H_i (for $0 \le i \le t_x$), M_i is defined as in Algorithm 11 (see alternate description in Algorithm 19 on page 30).

H_{t^*+1}: In hybrid H_{t^*+1}, M_i is defined as in H_{t^*}, and x_{t^*+1} is defined as $\mathsf{OEncode}(\epsilon^S)$, where ϵ^S just denotes a memory configuration in which every address is mapped to ϵ.

Lemma 6.5. *For all i with $0 \le i < t^*$, $H_i \approx H_{i+1}$.*

Proof. We give a sequence of indistinguishable hybrid distributions $H_i \approx H_{i,1} \approx \cdots \approx H_{i,5} \approx H_{i+1}$. Each hybrid $H_{i,j}$ is defined as $\mathsf{Garble}'(M_{i,j}, x_i)$, where $M_{i,j}$ is to be defined.

The indistinguishability of these hybrids is given in Claims 6.5.1 through 6.5.6.

Data: Puncturable PRF F, puncturable PRF G, transition function C, $y = M(x)$

Read q_{ORAM} from addresses $0, \ldots, \ell_q - 1$;
$\mathsf{out} := (q_0, \mathsf{Read}(0))$;
for $j = 0, \ldots, t^* - i - 1$ **do**
  Parse out as (q, op);
  Run $(q_{ORAM}, s) \leftarrow \mathsf{OAccess}(q_{ORAM}, \mathsf{op}; F(j,0), \ldots, F(j,\ell_r))$, but modify the generated external memory operations by adding ℓ_q to the accessed addresses;
  $\mathsf{out} \leftarrow C(q, s)$;
end
for $j = t^* - i, \ldots, t^*$ **do**
  for *each address a given by* $\mathsf{OSample}(j; G(j))$ **do**
    $\mathsf{Write}(a + \ell_q \mapsto 0)$ `// the value 0 is arbitrary`
  end
end
return y

Algorithm 11: Hybrid RAM machine M_i. See alternative description in Algorithm 19.

$H_{i,1}$**:** In hybrid $H_{i,1}$, $M_{i,1}$ is defined as in Algorithm 12 (or alternately in Algorithm 20), with p, $\iota_1, \ldots, \iota_p$ and $b_1, \ldots, b_p$ defined as follows.

Consider the underlying accesses defined by M on x. Let $A = (\mathbf{a}_1 | \cdots | \mathbf{a}_t)$ denote the physical addresses accessed by the ORAM when the randomness is generated by F. We define $\iota_1, \ldots, \iota_p$ as indices such that $\mathbf{a}_{t^*-i}$ depends on exactly $F(\iota_1), \ldots, F(\iota_p)$ (these indices are guaranteed to exist by Property 6.1). For each $k = 1, \ldots, p$, b_k is defined as $F(\iota_k)$.

$H_{i,2}$**:** In hybrid $H_{i,2}$, each b_k in $M_{i,2}$ is hard-coded as an independently and uniformly randomly chosen bit instead of as $F(\iota_k)$, and $\mathbf{a}_{t_x-i}$ is generated accordingly. That is, Property 6.1 says that $\mathbf{a}_{t_x-i}$ is a function of $F(\iota_1), \ldots, F(\iota_p)$. Here we generate $\mathbf{a}_{t_x-i}$ by applying that function to $b_1, \ldots, b_p$.

$H_{i,3}$**:** In hybrid $H_{i,3}$, $M_{i,3}$ is defined as in Algorithm 13 (alternate description in Algorithm 21, page 31). By observation, the locations accessed at time i depend only on $b_1, \ldots, b_p$, and are thusly hard-coded. The hard-coded mappings $\iota_k \mapsto b_k$ are removed, and F is unpunctured. Note now that $M_{i,3}$ is related to $b_1, \ldots, b_p$ only via $\mathbf{a}_{t_x-i}$.

$H_{i,4}$**:** In hybrid $H_{i,4}$, $\mathbf{a}_{t_x-i}$ is sampled as $\mathsf{OSample}(t_x - i)$ instead of using the implicit function of Property 6.1.

$H_{i,5}$**:** In hybrid $H_{i,5}$, $\mathbf{a}_{t_x-i}$ is sampled as $\mathsf{OSample}(t_x - i; G(t_x - i))$, instead of using true randomness in $\mathsf{OSample}$.

Claim 6.5.1. *For all i with $0 \le i < t^*$, $H_i \approx H_{i,1}$*

Proof. This follows from fixed-access security (in fact $i\mathcal{O}$ is directly applicable) because the transition functions of M_i and $M_{i,1}$ are functionally equivalent. Indeed, we are just replacing the sub-circuits for F and G with functionally equivalent sub-circuits. □

Claim 6.5.2. *For all i with $0 \le i < t^*$, $H_{i,1} \approx H_{i,2}$*

Proof. This follows from the pseudorandomness of F at the (selectively) punctured points. Given an adversary $\mathcal{A}$ which distinguishes $H_{i,1}$ from $H_{i,2}$, we construct an adversary $\mathcal{B}$ with non-negligible advantage in the puncturable PRF game.

Data: physical addresses $\mathbf{a}_{t_x-i}$, punctured PRF $F' = F\{\iota_1, \ldots, \iota_p\}$, indices $\iota_1, \ldots, \iota_p$, bits $b_1, \ldots, b_p$, punctured PRF $G' = G\{t_x - i\}$, $y = M(x)$

Read q_{ORAM} from addresses $0, \ldots, \ell_q - 1$, and treat it as an ORAM local state;
$\mathsf{out} := (q_0, \mathsf{Read}(0))$;
for $j = 0, \ldots, t_x - i - 1$ **do**
  Parse out as (q, op);
  Run $(q_{ORAM}, s) \leftarrow \mathsf{OAccess}(q_{ORAM}, \mathsf{op}; F'(j,0), \ldots, F'(j,\ell_r))$, but modify the generated external memory operations by adding ℓ_q to the accessed addresses. If any one of $(j,0), \ldots, (j,\ell_r - 1)$ is ι_k for some k, use b_k for randomness instead of evaluating F';
  $\mathsf{out} \leftarrow C(q, s)$;
  Run one step of M via the ORAM, using $r_j = (F'(j,0), \ldots, F'(j,\ell_r - 1))$ as randomness. If any one of $(j,0), \ldots, (j,\ell_r - 1)$ is ι_k for some k, use b_k instead of evaluating F'.
end
for *each address* a *in* $\mathbf{a}_{t_x-i}$ **do**
  $\mathsf{Write}(a \mapsto 0)$;
end
for $j = t_x - i + 1, \ldots, t_x$ **do**
  for *each address* a *given by* $\mathsf{OSample}(j; G(j))$ **do**
    $\mathsf{Write}(a \mapsto 0)$;
  end
end
return y

Algorithm 12: The hybrid RAM machine $M_{i,1}$. See alternative description in Algorithm 20.

Data: physical addresses $\mathbf{a}_{t_x-i}$, puncturable PRF F, punctured PRF $G' = G\{t_x - i\}$, $y = M(x)$

Read q_{ORAM} from addresses $0, \ldots, \ell_q - 1$;
$\mathsf{out} := (q_0, \mathsf{Read}(0))$;
for $j = 0, \ldots, t_x - i - 1$ **do**
  Parse out as (q, op);
  Run $(q_{ORAM}, s) \leftarrow \mathsf{OAccess}(q_{ORAM}, \mathsf{op}; F(j,0), \ldots, F(j,\ell_r))$, adding ℓ_q to all accessed addresses.;
  $\mathsf{out} \leftarrow C(q, s)$;
  Run one step of M via the ORAM, using $r_j = (F(j,0), \ldots, F(j,\ell_r - 1))$ as randomness.
end
for *each address* a *in* $\mathbf{a}_{t_x-i}$ **do**
  $\mathsf{Write}(a \mapsto 0)$;
end
for $j = t_x - i + 1, \ldots, t_x$ **do**
  for *each address* a *given by* $\mathsf{OSample}(j; G(j))$ **do**
    $\mathsf{Write}(a \mapsto 0)$;
  end
end
return y

Algorithm 13: The hybrid RAM machine $M_{i,3}$. See alternative description in Algorithm 21.

$\mathcal{B}$ requests a key K' punctured at $\iota_1, \ldots, \iota_p$ from the puncturable PRF challenger. Recall that Property 6.1 requires that $\iota_1, \ldots, \iota_p$ are efficiently computable given M and x.

$\mathcal{B}$ next receives a challenge $(b_1, \ldots, b_p)$ from the puncturable PRF challenger. Either every $b_i = F(\iota_i)$, or every b_i is sampled independently and uniformly at random from $\{0,1\}$. $\mathcal{B}$ generates $\mathbf{a}_{t_x - i}$ from $b_1, \ldots, b_p$ using the function given in Property 6.1.

$\mathcal{B}$ constructs a RAM machine M as in Algorithm 12, and generates $x' \leftarrow \mathsf{OEncode}(x, S, 1^\lambda)$. $\mathcal{B}$ sends $\mathsf{Garble}'(M, x'')$ to $\mathcal{A}$, where x'' is a memory configuration whose first ℓ_q bits map to q_{ORAM}, and the rest map to x'. Finally, $\mathcal{B}$ outputs whatever $\mathcal{A}$ returns. □

Claim 6.5.3. *For all i with $0 \le i < t^*$, $H_{i,2} \approx H_{i,3}$*

Proof. This follows from the fixed-access security of Garble' because the RAM machines $M_{i,2}$ and $M_{i,3}$ access the same sequence of addresses when given x_i as input. Indeed, Property 6.1 implies that changing the values of $b_1, \ldots, b_p$ cannot possibly change any of the accessed addresses other than at time $t_x - i$. But at time $t_x - i$, the accessed addresses are hard-coded and thus the same in both $M_{i,2}$ and $M_{i,3}$. □

Claim 6.5.4. *For all i with $0 \le i < t^*$, $H_{i,3} \approx H_{i,4}$*

Proof. This follows from the definition of $\mathsf{OSample}(i)$ in Property 6.2. Indeed, given $\mathbf{a}_{t_x - i}$ which is generated either as in an honest ORAM execution, or from $\mathsf{OSample}(i)$, an algorithm $\mathcal{B}$ can accordingly generate the distribution $H_{i,3}$ or $H_{i,4}$. □

Claim 6.5.5. *For all i with $0 \le i < t^*$, $H_{i,4} \approx H_{i,5}$*

Proof. This follows from the pseudorandomness of G at (selectively) punctured points. Given an adversary $\mathcal{A}$ which distinguishes $H_{i,4}$ from $H_{i,5}$, we construct an adversary $\mathcal{B}$ with non-negligible advantage in the puncturable PRF game.

$\mathcal{B}$ requests G' punctured at $t_x - i$ from the puncturable PRF challenger. Note that t_x is computable from M, x, and the worst-case time bound T.

$\mathcal{B}$ next receives a challenge r from the puncturable PRF challenger, such that r is either $G(t_x - i)$ or r is sampled uniformly at random. B generates $\mathbf{a}_{t_x - i} \leftarrow \mathsf{OSample}(t_x - i;\ r)$.

$\mathcal{B}$ samples the puncturable PRF F, samples $q_{ORAM}, x' \leftarrow \mathsf{OEncode}(x)$, and sends $\mathsf{Garble}'(M, x'')$ to $\mathcal{A}$, where M is described in Algorithm 13 and x'' is a memory configuration whose first ℓ_q bits map to q_{ORAM}, and the rest map to x'. Finally, $\mathcal{B}$ outputs whatever $\mathcal{A}$ outputs. □

Claim 6.5.6. *For all i with $0 \le i < t^*$, $H_{i,5} \approx H_{i+1}$*

Proof. This follows from the fixed-access security of Garble' because $H_{i,5}$ and H_{i+1} access the same sequence of locations. Indeed, the only difference between $H_{i,5}$ and H_{i+1} is that a loop in H_{i+1} is partially unrolled in $H_{i,5}$.

□

This concludes the proof of Lemma 6.5. □

Lemma 6.6. $H_{t^*} \approx H_{t^*+1}$

Proof. This follows from the fixed-access security of Garble'. Both M_{t^*} and M_{t^*+1} access the same sequence of addresses independently of their inputs, and x' and x_{t^*+1} have the same sets of non-empty addresses. □

□

References

[AB15] Benny Applebaum and Zvika Brakerski. Obfuscating circuits via composite-order graded encoding. In Dodis and Nielsen [DN15], pages 528–556.

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, pages 52–73, 2014.

[BGK+14] Boaz Barak, Sanjam Garg, YaelTauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In PhongQ. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 221–238. Springer Berlin Heidelberg, 2014.

[BGL+15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for ram programs. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015.

[CLTV15] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Probabilistic indistinguishability obfuscation. In *TCC*, 2015.

[CP13] Kai-Min Chung and Rafael Pass. A simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.

[DN15] Yevgeniy Dodis and Jesper Buus Nielsen, editors. *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, volume 9015 of *Lecture Notes in Computer Science*. Springer, 2015.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability and functional encryption for all circuits. In *FOCS*, pages 40–49, 2013.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, pages 405–422, 2014.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564, 2013.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled ram from one-way functions. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015.

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

[HW14] Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. Cryptology ePrint Archive, Report 2014/669, 2014. http://eprint.iacr.org/.

[IPS15] Yuval Ishai, Omkant Pandey, and Amit Sahai. Public-coin differing-inputs obfuscation and its applications. In Dodis and Nielsen [DN15], pages 668–697.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer Berlin Heidelberg, 2013.

[Mer88] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Advances in CryptologyCRYPTO87*, pages 369–378. Springer, 1988.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with o((logn)3) worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC*, pages 475–484, 2014.

[Yao82] Andrew C Yao. Protocols for secure computations. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE, 1982.

[Zim14] Joe Zimmerman. How to obfuscate programs directly. *IACR Cryptology ePrint Archive*, 2014:776, 2014.

A ORAM Construction

We describe the ORAM of [CP13] (which is a simplification of [SCSL11]) for an underlying memory $x : \mathbb{N} \to \Sigma$ of size S. We describe a solution which requires $S/2$ registers. In order to use a constant or polylogarithmic number of registers, we recursively replace the $S/2$ registers with an $S/2$-symbol ORAM unless S is sufficiently small.

We first describe the memory layout of the $S/2$ register ORAM, and then we describe the procedure $\mathsf{OAccess}$ for emulating accesses to the S-symbol memory. For the sake of our recursion, we define $\mathsf{OAccess}$ to have a slightly more general interface than an ordinary RAM memory. In particular, $\mathsf{OAccess}$ can process a memory operation which reads and writes simultaneously. Such an operation is represented as an address $a \in \mathbb{N}$ and a update function $u : \Sigma \to \Sigma$, where Σ is the alphabet of the emulated memory.

Memory Layout Memory is laid out as a complete binary tree with $S/2$ leaves. Each node of the tree is a bucket which can hold up to $\log^2(\lambda)$ tuples of the form $(b, \mathsf{pos}, s_0, s_1)$. Here $b \in [S/2]$ is a block identifier, $\mathsf{pos} \in [S/2]$ identifies a leaf of the binary tree, and s_0 and s_1 are values of underlying memory corresponding to block b.

These tuples store the data of the S-symbol memory x in blocks of size 2. A block b stores the data of two underlying adjacent addresses $2b$ and $2b+1$. Each block b is "assigned" to a a leaf pos_b of the tree in memory. The mapping Pos from b to pos_b is maintained in the $S/2$ registers.

The main invariant of the ORAM is that if b is assigned to pos_b, then $(b, \mathsf{pos}_b, x(2b), x(2b+1))$ is stored in the tree somewhere along the path from the root to pos_b.

$\mathsf{OAccess}$ $\mathsf{OAccess}(a, u)$ takes an address $a \in [S]$ and an update function $u : \Sigma \to \Sigma$. $\mathsf{OAccess}$ then accesses memory several times (in fact $\mathrm{poly}(\log S)$ times), before returning a symbol $s \in \Sigma$. Informally, $\mathsf{OAccess}(a, u)$ proceeds in three stages:

1. Look up $\mathsf{pos}_b \leftarrow \mathsf{Pos}(b)$, where b is the block $\lfloor a/2 \rfloor$. Access each bucket on the path to pos_b. If a tuple of the form $(b, \mathsf{pos}_b, s_0, s_1)$ is found, remember the values s_0 and s_1. Otherwise, define $s_0 = s_1 = \epsilon$.
2. We pick a new random leaf pos'_b, and set $\mathsf{Pos}(b) \leftarrow \mathsf{pos}'$.

3. We write $(b, \mathsf{pos}'_b, u(s_0), s_1)$ to the root bucket if a is even, and write $(b, \mathsf{pos}'_b, s_0, u(s_1))$ otherwise.

4. We pick a random leaf pos^*, and "push" tuples down the path from the root to pos^* such that after this traversal, the following property holds. Each tuple of the form $(\star, \mathsf{pos}, \star, \star)$ (for any pos) on this path is located at the least common ancestor of pos and pos^*.

Formally, our implementation of $\mathsf{OAccess}$ is described in Algorithm 14. We use the notation that $\mathcal{P}(\mathsf{pos})$ denotes the path in the tree from the root to pos.

Input: address a, update function u
Data: Pointer to a smaller ORAM Pos
Choose a uniformly random leaf pos';
$\mathsf{pos} \leftarrow \mathsf{Pos}.\mathsf{OAccess}(\lfloor \frac{a}{2} \rfloor, (\cdot \mapsto \mathsf{pos}'))$;
$s \leftarrow \texttt{AccessPath}(\mathcal{P}(\mathsf{pos}), a, u, \mathsf{pos}')$;
$\texttt{Flush}()$;
return v;

Algorithm 14: Recursive $\mathsf{OAccess}$

Input: address a, update function u
Data: Pointer to an array A
foreach *entry in* A **do**
  If an entry is of the form (a, v), write $(a, u(v))$ in its place and save v;
  Otherwise, write the entry back unchanged;
end
return v;

Algorithm 15: $\mathsf{OAccess}$ Base Case

Input: path P, address a, update function u, new leaf pos'
foreach *row of each bucket on path* P **do**
  If the row is of the form $(\lfloor a/2 \rfloor, \mathsf{pos}, v_0, v_1)$, then write ϵ in its place;
  Otherwise, write back the row unchanged;
end
if a *is even* **then**
  Write $(\lfloor a/2 \rfloor, \mathsf{pos}', u(v_0), v_1)$ to the root bucket;
  return v_0;
else
  Write $(\lfloor a/2 \rfloor, \mathsf{pos}', v_0, u(v_1))$ to the root bucket;
  return v_1;
end

Algorithm 16: AccessPath

A.1 Localized Randomness

As per the definition in Section 2, we show that this ORAM satisfies two properties. The first property is that for a given underlying sequence of memory acces instructions, the actual physical addresses accessed by the ORAM scheme for each operation depend on small, disjoint subsets of the random bits used.

Indeed, when $\mathsf{OAccess}$ is sequentially invoked on $\mathsf{op}_1, \ldots, \mathsf{op}_t$ (corresponding to addresses $a_1, \ldots, a_t$), the addresses $\mathbf{a}_i$ accessed in the emulation of op_i consist of two parts:

Input: None
Pick a random leaf pos^*;
Initialize an empty list L;
for *each bucket* B *on the path to* pos^* **do**
    Add all tuples in L to B;
    Reset L to be the empty list;
    for *each tuple* $(b, \mathsf{pos}, v_0, v_1)$ *in* B **do**
        If B is not the least common ancestor of pos and pos^*, then delete $(b, \mathsf{pos}, v_0, v_1)$ from B, and add it to L.
    end
end

Algorithm 17: Flush

- The addresses accessed in "fetching", i.e. the path from root to $\mathsf{Pos}(\lfloor a_i/2 \rfloor)$.
- The addresses accessed in "flushing", i.e. the path from the root to pos^*.

The path from root to $\mathsf{Pos}(\lfloor a_i/2 \rfloor)$ is sampled when the value of $\mathsf{Pos}(\lfloor a_i/2 \rfloor)$ was determined. This happens on the last access before i of block $\lfloor a_i/2 \rfloor$. Since blocks are disjoint, the randomness used here is also disjoint from the randomness determining any other $\mathbf{a}_j$ for $j \neq i$.

The leaf pos^* is sampled at time i, so the flushing randomness used for $\mathbf{a}_i$ is obviously disjoint from the randomness determining $\mathbf{a}_j$ for $i \neq j$.

The final property we used was the existence of an efficient $\mathsf{OSample}$ algorithm which samples $\mathbf{a}_i$ (and in particular depends only polylogarithmically on i). This is easy – one can simply output two paths from the root to independently random leaves.

B Alternate Algorithm Descriptions

This section provides alternative presentations of the garbling scheme and the hybrid algorithms in Section 6. This presentation emphasizes the reactive nature of these algorithms and is closer to the syntax of RAM machines as defined in Section 2. It is stressed that the algorithm themselves are meant to be identical, once translated to some fixed programming language. it is only the presentation to the reader that differs.

Data: transition function C, initial state q_0, Puncturable PRF F, initial state q_0^{ORAM} for $\mathsf{OAccess}$.
In first activation:
    $(q, s, a, \mathsf{op}) \leftarrow (q_0, \perp, 0, \mathsf{Read})$; $t \leftarrow 0$;
    activate $\mathsf{OAccess}(q_0^{ORAM}, a, s, \mathsf{op}, t, F(t,0), \ldots, F(t, \ell_r))$;
On input (q^{ORAM}, s) **from** $\mathsf{OAccess}$ **do:**
    $\mathsf{out} \leftarrow C(q, s)$;
    If $\mathsf{out} \in Y$ **then** output out and halt;
    parse $q, a, s, \mathsf{op} \leftarrow \mathsf{out}$;
    $t \leftarrow t + 1$;
    activate $\mathsf{OAccess}(q^{ORAM}, a, s, \mathsf{op}, t, F(t,0), \ldots, F(t, \ell_r))$;

Algorithm 18: Definition of the garbled machine M'. This is alternative presentation to Algorithm 10

Data: transition function C, initial state q_0, Puncturable PRFs F, G, initial state q_0^{ORAM} for $\mathsf{OAccess}$, $y = M(x)$, time bounds i, t_x.

In first activation:
  $(q, s, a, \mathsf{op}) \leftarrow (q_0, \perp, 0, \mathsf{Read})$; $j \leftarrow 0$;
  activate $\mathsf{OAccess}(q_0^{ORAM}, a, s, \mathsf{op}, t, F(j,0), \ldots, F(j,\ell_r))$;
On input (q^{ORAM}, s) **from** $\mathsf{OAccess}$ **do:**
  If $j = t_x$ **then** output y and halt;
  $\mathsf{out} \leftarrow C(q, s)$;
  parse $q, a, s, \mathsf{op} \leftarrow \mathsf{out}$;
  $j \leftarrow j + 1$;
  If $j < i$ **then** activate $\mathsf{OAccess}(q^{ORAM}, a, s, \mathsf{op}, i, F(i,0), \ldots, F(i,\ell_r))$;
  If $j \geq i$ **then**
    run $\mathbf{a} \leftarrow \mathsf{OSample}(j, F(j,0), \ldots, F(j,\ell_r))$;
    for $k = 1..|\mathbf{a}|$, in the kth activation from now, output $(\mathsf{Write}, a_k, 0)$ where a_k is the kth address in $\mathbf{a}$ (without incrementing j);

Algorithm 19: Hybrid RAM machine M_i. This is an alternative dscription to Algorithm 11.

Data: transition function C, initial state q_0, initial state q_0^{ORAM} for $\mathsf{OAccess}$, $y = M(x)$, time bounds i, t_x, physical addresses $\mathbf{a}_{t_x - i}$, punctured PRF $F' = F\{\iota_1, \ldots, \iota_p\}$, indices $\iota_1, \ldots, \iota_p$, bits $b_1, \ldots, b_p$, punctured PRF $G' = G\{t_x - i\}$.

In first activation:
  $(q, s, a, \mathsf{op}) \leftarrow (q_0, \perp, 0, \mathsf{Read})$; $j \leftarrow 0$;
  activate $\mathsf{OAccess}(q_0^{ORAM}, a, s, \mathsf{op}, t, F(j,0), \ldots, F(j,\ell_r))$;
On input (q^{ORAM}, s) **from** $\mathsf{OAccess}$ **do:**
  If $j = t_x$ **then** output y and halt;
  $\mathsf{out} \leftarrow C(q, s)$;
  parse $q, a, s, \mathsf{op} \leftarrow \mathsf{out}$;
  $j \leftarrow j + 1$;
  If $j < i$ **then** activate $\mathsf{OAccess}(q^{ORAM}, a, s, \mathsf{op}, i, F(i,0), \ldots, F(i,\ell_r))$, where:
  If any one of $(t,0), \ldots, (t,\ell_r - 1)$ is ι_k for some k, use b_k instead of evaluating F';
  If $j = i$ **then** for $k = 1..|\mathbf{a}_{t_x - i}|$, in the k^{th} activation from now, output $(\mathsf{Write}, a_k, 0)$ whre a_i is the kth address in $\mathbf{a}_{t_x - i}$ (without incrementing j);
  If $j > i$ **then**
    run $\mathbf{a} \leftarrow \mathsf{OSample}(j, F(j,0), \ldots, F(j,\ell_r))$;
    for $k = 1..|\mathbf{a}|$, in the kth activation from now, output $(\mathsf{Write}, a_k, 0)$ where a_k is the kth address in $\mathbf{a}$ (without incrementing j);

Algorithm 20: The hybrid RAM machine $M_{i,1}$. This is alternative description to Algorithm 12.

Data: transition function C, initial state q_0, initial state q_0^{ORAM} for $\mathsf{OAccess}$, $y = M(x)$, time bounds i, t_x, physical addresses $\mathbf{a}_{t_x - i}$, puncturable PRF F, punctured PRF $G' = G\{t_x - i\}$, $y = M(x)$

In first activation:
  $(q, s, a, \mathsf{op}) \leftarrow (q_0, \perp, 0, \mathsf{Read})$; $j \leftarrow 0$;
  activate $\mathsf{OAccess}(q_0^{ORAM}, a, s, \mathsf{op}, t, F(j,0), \ldots, F(j, \ell_r))$;
On input (q^{ORAM}, s) **from** $\mathsf{OAccess}$ **do:**
  If $j = t_x$ **then** output y and halt;
  $\mathsf{out} \leftarrow C(q, s)$;
  parse $q, a, s, \mathsf{op} \leftarrow \mathsf{out}$;
  $j \leftarrow j + 1$;
  If $j < i$ **then** activate $\mathsf{OAccess}(q^{ORAM}, a, s, \mathsf{op}, i, F(i,0), \ldots, F(i, \ell_r))$;
  If $j = i$ **then** for $k = 1..|\mathbf{a}_{t_x - i}|$, in the k^{th} activation from now, output $(\mathsf{Write}, a_k, 0)$ whre a_i is the kth address in $\mathbf{a}_{t_x - i}$ (without incrementing j);
  If $j > i$ **then**
    run $\mathbf{a} \leftarrow \mathsf{OSample}(j, F(j,0), \ldots, F(j, \ell_r))$;
    for $k = 1..|\mathbf{a}|$, in the k^{th} activation from now, output $(\mathsf{Write}, a_k, 0)$ where a_k is the kth address in $\mathbf{a}$ (without incrementing j);

Algorithm 21: The hybrid RAM machine $M_{i,3}$. This is alternative description to Algorithm 13.