Fully Succinct Garbled RAM

Ran Canetti*  Justin Holmgren†

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.

In particular, this opens the door to obfuscated computations that are sublinear in the length of their inputs.

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 in part by NSF Frontier CNS1413920, the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army Research Office under contract number W911NF-15-C-0226
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 [Yao86], 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 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 et al. and Garg 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 et al. and Canetti 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 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. 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.

Furthermore, similarly to the schemes of [CHJV15, BGL+15, KLW15], our garbling scheme supports persistent data: Multiple machines $M_1, \ldots, M_\ell$ can be garbled, along with some (potentially very large) data, such that machine $M_i$ acts on the data configuration left by $M_{i-1}$, and such that having access to the garbled data and garbled machines gives no information other than $y_1, \ldots, y_\ell$, where $y_i$ is the output of $M_i$ when executed in sequence on the data after $M_1, \ldots, M_{i-1}$. Importantly, in our case of RAM machines, each machine can run in time that is sublinear in the entire data; for example, each machine may execute a database query, modifying the database and returning some small result. Our transformation preserves the sub-linear complexity of the machines.

The preservation of sublinear complexity is powerful also in the context of delegation of computation. Indeed, consider the task of delegating the computation of sublinear-time RAM programs over large delegated databases. Indeed, when instantiated with our scheme, the delegation of computation scheme for RAM-IO (described in [CHJV15]) is the first to guarantee both correctness and privacy of the computation, while preserving full succinctness and sublinear complexity for the prover.\footnote{In particular, the “generic” method of guaranteeing privacy in delegation schemes by encrypting the data using fully homomorphic encryption incurs a super-linear complexity overhead. We thank Yael Kalai for this observation.}

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, naive attempts to extend the techniques of [KLW15] to RAM programs encounter the following problem: The [KLW15] technique applies when the plaintext machine is deterministic and its memory access pattern is fixed and independent of the inputs. When the plaintext program is a Turing machine, making sure that the memory access pattern is fixed incurs only small overhead in complexity. 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, this is the case for Oblivious RAM schemes [GO96]. 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. However, 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 [CHJV15] technique (which is ORAM-friendly) and the [KLW15] technique (which is not). 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 $i$th 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, this new structure of the hybrid programs 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 at steps \( i \) and up is random even given the program’s state at steps 1 through \( i - 1 \). 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, it seems likely 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 as 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 any same-length input values \( x_0 \) and \( x_1 \), it holds that \( (\hat{M}_0, \hat{x}_0) \approx (\hat{M}_1, \hat{x}_1) \) as long as (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 \( x_1 \). (Here \( \hat{M} \) and \( \hat{x} \) are the garbled versions of \( M \) and \( x \), respectively.) Furthermore, the sequence of addresses accessed by \( \hat{M} \) on input \( \hat{x} \) is identical to the sequence of addresses accessed by \( M \) on input \( x \).

The fact that \( \hat{M} \) preserves the access pattern of \( M \) 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 \( (\hat{M}_0, \hat{x}_0) \approx (\hat{M}_1, \hat{x}_1) \) 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 [KLV15] 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 [KLV15] 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: 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. 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 $M'$ 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 $(\hat{M}_0', x_0) \approx (\hat{M}_1', x_1)$, we consider an intermediate hybrid in which $M_0'$ is replaced by a new machine $M_0$ which now executes $M_0$ on track ‘A’ but $M_1$ on track ‘B’. Indistinguishability of the intermediate hybrid from either end is shown by demonstrating how to indistinguishably switch from a machine that outputs the result of track ‘A’ to a machine that outputs the result of track ‘B’.

### 1.1.2 Full Garbling

Our final and main step is a construction of a succinct fully secure garbler for RAM machines from a succinct fixed-address garbler. 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.

Recall that for a fully secure garbler we require that $(\hat{M}_0', x_0) \approx (\hat{M}_1', x_1)$ whenever $M_0(x_0) = M_1(x_1)$, and in addition the runtime and space requirement of $M_0$ on $x_0$ is the same as the runtime and space requirement of $M_1$ on $x_1$.

Furthermore, recall that hiding the memory access pattern in an efficiency preserving way is done by Oblivious RAM (ORAM) techniques, which make crucial use of randomness that remains secret within the program. In contrast, our fixed-address garbler guarantees security only when the access pattern of the underlying machine is fixed.

Our first step towards making use of a fixed-address garbler is to “derandomize” the ORAM scheme by setting its randomness to be the result of applying a puncturable PRF to the program’s input. This indeed means that, for any given input, the access pattern is fixed. Still, it is not clear how to argue security of the scheme; in particular, the access pattern of $\hat{M}_0$ when run on $\hat{x}_0$ may well be different than the access pattern of $\hat{M}_1$ when run on $\hat{x}_1$.

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 $a_1, \ldots, a_t$ that it accesses. Here each $a_i$ is itself a sequence of addresses $a_{i,1}, \ldots, a_{i,n}$, accessed in the emulation of the underlying RAM machine’s $i^{th}$ step. Specifically, we require that (for given underlying memory operations $o_{p_1}, \ldots, o_{p_t}$), each $a_i$ is influenced only by a small subset $D_i$ of the bits of $R$, and each bit of $R$ influences at most one of $a_i$. The ORAM must also come with a p.p.t. algorithm $O\text{Sample}$ such that $O\text{Sample}(i)$ has the same distribution as $a_i$, independently of $o_{p_1}, \ldots, o_{p_t}$. A simple analysis in Appendix A shows that the ORAM of Chung and Pass [CP13, SCS11] 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 program that for each $i$ from 1 to $T$, simulates addresses $a_i$ to access using $\text{Sim}(F(i))$ for some puncturable PRF $F$, and output the resulting garbled program. We need
to prove that this simulation is indistinguishable from the real garbled machine $M$, in a sequence of hybrids which changes each $a_i$ to $\text{Sim}(F(i))$.

This argument is reminiscent of the proof of security for the [CLTV15] construction of a probabilistic iO (PIO) obfuscator, with the complication that $a_1$ through $a_t$ are generated adaptively. This complication is handled by switching the $a_i$'s in reverse order – starting with $a_t$ and ending with $a_1$. Here it is crucial to note that, despite the adaptivity, $a_1$ through $a_t$ are mutually independent random variables by the localized randomness property of the ORAM scheme.

To switch $a_i$ to $\text{Sim}(F(i))$, we first hard-code $a_i$, and then puncture the ORAM’s PRF on exactly the points which determine $a_i$. ORAM locality implies that this set is small and that the puncturing does not affect any $a_j$ for $j \neq i$, so the security of the fixed access garbler is applicable. We then indistinguishably replace $a_i$ with $\text{Sim}(F(i))$. Finally we remove the hard-codings and unpuncture all the PRFs, relying again on security of the fixed-access garbler.

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. Section 7 presents the application to persistent data.

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 Notation

- $\mathbb{N}$ denotes the set $\{0, 1, 2, \ldots\}$. For any integer $n \in \mathbb{N}$, $[n]$ denotes the set $\{0, 1, \ldots, n - 1\}$.
- For a set $X$ and a set $Y$, $Y^X$ denotes the set of all functions from $X$ to $Y$. When $X = \mathbb{N}$, $f \in Y^\mathbb{N}$ is also identified as the infinite sequence $(f(0), f(1), \ldots)$.
- For $n \in \mathbb{N}$, $X^n$ denotes the set of $n$-tuples of elements of $X$.
- $X^*$ denotes $\cup_{i \in \mathbb{N}} X^i$.
- For a set $S \subset [n]$, $S = \{i_1, \ldots, i_t\}$ with $i_1 < \cdots < i_t$, and a sequence $\vec{a} = (a_0, \ldots, a_{n - 1}) \in X^n$, we write $\vec{a}_S$ to denote the tuple $(a_{i_1}, \ldots, a_{i_t})$. We use analogous notation for subsequences of infinite sequences $(X^\mathbb{N})$. More generally, if $f$ is a function from $X$ to $Y$, and if $S$ is a subset of $X$, we write $f(S)$ to denote $\{f(x) : x \in S\}$. If $S$ is an ordered set, $f(S)$ inherits the same ordering.
- For a finite set $S$, we write $\ell_S$ to denote the worst-case length of binary strings encoding elements of $S$ (typically this will be $\lceil \log(|S|) \rceil$). We identify $S$ with a subset of $\{0, 1\}^{\ell_S}$.
- For a randomized algorithm $A$, we write $A(x; r)$ to denote running $A$ on input $x$ with randomness $r$. 

5
2.2 Indistinguishability Obfuscation

We assume the existence of an indistinguishability obfuscator \([BGI^{+}01, GGH^{+}13]\).

**Syntax.** An indistinguishability obfuscator for circuits is a p.p.t. algorithm \(iO\) which takes as input a security parameter \(1^\lambda\), a circuit \(C\), and outputs a circuit \(\tilde{C}\).

**Correctness.** For all \(x\), \(\Pr[iO(1^\lambda, C)(x) = C(x)] = 1\).

**Security.** If \(|C_0| = |C_1|\) and \(C_0(x) = C_1(x)\) for every \(x\), then \(iO(1^\lambda, C_0) \approx iO(1^\lambda, C_1)\).

2.3 The RAM Model

2.3.1 RAM Machines

In this work, a RAM machine \(M\) is defined as a tuple \((\Sigma, Q, Y, C)\), where:

- \(\Sigma\) is a finite set, which is the possible contents of a memory cell. For example, \(\Sigma = \{0, 1\}\).
- \(Q\) 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 \(Q \times (\Sigma \cup \{\epsilon\}) \rightarrow (Q \times O_\Sigma) \cup Y\). Here \(O_\Sigma\) denotes the set of memory operations with \(\Sigma\) as the alphabet of possible memory symbols. Precisely, \(O_\Sigma = (\mathbb{N} \times \Sigma)\). 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 write \(|M|\) to denote the tuple \((\ell_\Sigma, \ell_Q, \ell_Y, |C|)\), where \(\ell_\Sigma\) is the length of a binary encoding of \(\Sigma\), and similarly for \(\ell_Q\) and \(\ell_Y\).

2.3.2 Memory Configurations

A memory configuration on alphabet \(\Sigma\) is a function \(s : \mathbb{N} \rightarrow \Sigma \cup \{\epsilon\}\). Let \(\|s\|_0\) denote \(|\{a : s(a) \neq \epsilon\}|\) and, in a horrific abuse of notation, let \(\|s\|_\infty\) denote \(\max(\{a : s(a) \neq \epsilon\})\), which we will call the length of the memory configuration. A memory configuration \(s\) can be implemented (say with a balanced binary tree) by a data structure of size \(O(\|s\|_0)\), supporting updates to any index in \(O(\log \|s\|_\infty)\) time.

We can naturally identify a string \(x = x_1 \ldots x_n \in \Sigma^*\) with the memory configuration \(s_x\), defined by

\[
s_x(i) = \begin{cases} x_i & \text{if } i \leq |x| \\ \epsilon & \text{otherwise} \end{cases}
\]

Looking ahead, efficient representations of sparse memory configurations (in which \(\|s\|_0 < \|s\|_\infty\)) are convenient for succinctly garbling computations where the space usage is larger than the input length.
2.3.3 Execution

We now define what it means to execute a RAM machine $M = (\Sigma, Q, Y, C)$ on an initial memory configuration $s_0 \in \Sigma^N$ to obtain $M(s_0)$.

Define $a_0 = 0$. For $i > 0$, iteratively define $(q_i, a_i, v_i) = C(q_{i-1}, s_{i-1}(a_{i-1}))$ and define the $i^{th}$ memory configuration $s_i$ as

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

If $C(q_{t-1}, s_{t-1}(a_{t-1})) = y \in Y$ for some $t$, then we say that $M(s_0) = y$. If there is no such $t$, we say that $M(s_0) = \bot$. When $M(s_0) \neq \bot$, it is convenient to define the following functions:

- Define the running time of $M$ on $s_0$ as the above $t$, and denote it $\text{Time}(M, s_0)$.
- Define the space usage of $M$ on $s_0$ as $\max_{i=0}^{t-1}(\|s_i\|_\infty)$, and denote it $\text{Space}(M, s_0)$.
- Define the execution transcript of $M$ on $s_0$ as $(q_0, a_0, v_0), \ldots, (q_{t-1}, a_{t-1}, v_{t-1}, y)$, and denote it $T(M, s_0)$.
- Define the resultant memory configuration of $M$ on $s_0$ as $s_t$, and denote it $\text{NextMem}(M, s_0)$.

2.4 Garbling

Syntax. A garbling scheme for RAM programs is a tuple of p.p.t. algorithms $(\text{KeyGen}, \text{GbPrg}, \text{GbMem}, \text{Exec})$.

- Key Generation: $\text{KeyGen}(1^\lambda, S, T)$ takes the security parameter $\lambda$ in unary, a space bound $S$ and a time bound $T$ in binary, and outputs a secret key $SK$.
- Machine Garbling: $\text{GbPrg}(SK, M)$ takes as input a secret key $SK$ and a RAM machine $M$, and outputs a RAM machine $\tilde{M}$.
- Memory Garbling: $\text{GbMem}(SK, s)$ takes as input a secret key $SK$ and a memory configuration $s$, and then outputs a memory configuration $\tilde{s}$.

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

Correctness. A garbling scheme is said to be correct if for all RAM machines $M$ and all memory configurations $s$ such that $\text{Time}(M, s) \leq T$ and $\text{Space}(M, s) \leq S$, we have

$$\Pr \left[ \tilde{M}(\tilde{s}) = M(s) \mid SK \leftarrow \text{KeyGen}(1^\lambda, S, T) \right. \left. \tilde{M} \leftarrow \text{GbPrg}(SK, M) \right. \tilde{s} \leftarrow \text{GbMem}(SK, s) \right] \geq 1 - \text{negl}(\lambda).$$

Efficiency. A garbling scheme is said to be efficient if:

1. $\text{KeyGen}$, $\text{GbPrg}$, and $\text{GbMem}$ are all probabilistic 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.

   - The running time of $\text{GbMem}$ is polynomial in $\|s\|_0$, the number of non-empty addresses in $s$. In fact in our scheme the running time is linear in $\|s\|_0$.

2. $\text{Time}(\tilde{M}, \tilde{s}) = \tilde{O}(\text{Time}(M, s))$ (hiding polylogarithmic factors in $S$), and $\text{Space}(\tilde{M}, \tilde{s}) \leq S$. 

7
Security. A garbling scheme is said to be secure if there is an efficient algorithm Sim such that for all RAM machines $M$ and memory configurations $s$ with $\text{Time}(M, s) \leq T$ and $\text{Space}(M, s) \leq S$, no p.p.t. algorithm can distinguish
\[
\begin{align*}
\hat{M}, \hat{s} & \leftarrow \text{KeyGen}(1^\lambda, S, T) \\
\hat{M} & \leftarrow \text{GbPrg}(SK, M) \\
\hat{s} & \leftarrow \text{GbMem}(SK, s)
\end{align*}
\]
from
\[
\text{Sim}(1^\lambda, M(s), \text{Time}(M, s), T, S, |M|, \|s\|_0).
\]

2.5 Cryptographic Iterators

Roughly speaking, a cryptographic iterator is a family of collision-resistant hash functions which is iO-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 $H(m_k || H(m_{k-1}) \cdots H(m_1 || 0^n))$, 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 specific 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 Setup-Itr to Setup and renaming Setup-Itr-Enforce to SetupEnforce. Formally, a cryptographic iterator for the message space $\mathcal{M} = \{0, 1\}^n$ consists of the following probabilistic polynomial-time algorithms. Setup and SetupEnforce are randomized algorithms, but 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.

$\text{Setup}(1^\lambda, T) \rightarrow \text{PP}, \text{itr}_0$

Setup takes as input the security parameter $\lambda$ in unary and a binary bound $T$ on the number of iterations. Setup then outputs public parameters $\text{PP}$ and an initial iterator value $\text{itr}_0$.

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

SetupEnforce takes as input the security parameter $\lambda$ 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 $k < T$. SetupEnforce then outputs public parameters $\text{PP}$ and an initial iterator value $\text{itr}_0$.

$\text{Iterate}(\text{PP}, \text{itr}_{in}, m) \rightarrow \text{itr}_{out}$

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

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

\[
\text{Iterate}^k(\text{PP}, \text{itr}, (m_1, \ldots, m_k)) = \text{Iterate}(\text{PP}, \text{Iterate}^{k-1}(\text{PP}, \text{itr}, (m_1, \ldots, m_{k-1})), m_k).
\]

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

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

Enforcing

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

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

2.6 Positional Accumulators

Positional accumulators (PAs) are an iO-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 $M^*$ 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. $\text{SetupAcc}$ and $\text{SetupAccEnforceUpdate}$ are randomized, while $\text{Update}$ and $\text{LocalUpdate}$ are deterministic. For a given memory configuration $x$, there is a uniquely defined accumulator value $ac_x$. The procedures $\text{Update}$ and $\text{LocalUpdate}$ allow for efficient local update and opening. The memory operations supported are of the form $\text{ReadWrite}(addr_i \mapsto v_i)$ for some address $addr_i$ and value $v_i$. This writes to memory, while returning the previous value stored at $addr_i$.

$\text{SetupAcc}(1^\lambda, S) \rightarrow PP, ac_0, store_0$

The setup algorithm takes as input the security parameter $\lambda$ in unary and a bound $S$ (in binary) on the memory addresses accessed. $\text{SetupAcc}$ 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.

$\text{Update}(PP, store_{in}, op) \rightarrow store_{out}, aux$

The prep-update algorithm takes as input the public parameters $PP$, data store $store_{in}$, and operation $op$. $\text{PrepUpdate}$ 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.

$\text{LocalUpdate}(PP, ac_{in}, op, aux) \rightarrow (v, ac_{out})$ 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 $aux$. $\text{LocalUpdate}$ then either outputs a value $v$ and a new accumulator value $ac_{out}$, or $\text{LocalUpdate}$ 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 $\text{LocalUpdate}$ other than the honestly generated one.

$\text{SetupAccEnforceUpdate}(1^\lambda, S, op_1, \ldots, op_k) \rightarrow PP, ac_0, store_0$

The alternate setup algorithm additionally take as inputs a sequence of memory operations $op_1, \ldots, op_k$ for some integer $k$. $\text{SetupAccEnforceUpdate}$ 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 $\text{SetupAcc}$ is that the output of $\text{SetupAccEnforceUpdate}$ satisfies an additional information theoretic “enforcing” property.

A positional accumulator must satisfy the following properties.

---

\[2\text{Technically for evaluation to be efficient, the input } store_{in}\text{ should be a pointer to a data store}\]
Correctness 
Let $op_0, \ldots, op_k$ be any arbitrary sequence of memory operations.

We first define the “correct” $v_i^*$ as follows. Say that $op_i$ accesses address $addr_i$. If no $op_j$ for $j < i$ accesses $addr_i$, then $v_i^*$ is $\epsilon$. Otherwise, let $j_i$ be the largest $j$ such that $j < i$ and $op_j$ accesses $addr_i$. We define $v_i^*$ such that $op_j$ is of the form $\text{ReadWrite}(addr_i \mapsto v_i^*)$.

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

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

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

Setup Indistinguishability
For any sequence of operations $op_0, \ldots, op_k$, any space bound $S$, and any p.p.t. algorithm $A$, setup indistinguishability requires that

$$\text{SetupAcc}(1^\lambda, S) \approx \text{SetupAccEnforceUpdate}(1^\lambda, S, op_0, \ldots, op_k)$$

Enforcing
Enforcing requires that for all space bounds $S$, all sequences of operations $op_1, \ldots, op_k$, and all $aux'$, we have

$$\Pr \left[ v \in \{(v_i^*, \text{ac}_{k+1}), \bot\} \middle| \begin{array}{l} \text{PP, ac}_0, \text{store}_0 \leftarrow \text{SetupAccEnforceUpdate}(1^\lambda, S, op_0, \ldots, op_k) \\ \text{For } i = 0, \ldots, k-1 \\ \text{store}_{i+1}, \text{aux}_i \leftarrow \text{Update}(\text{PP, store}_i, op_i) \\ (v_i, \text{ac}_{i+1}) \leftarrow \text{LocalUpdate}(\text{PP, ac}_i, op_i, \text{aux}_i) \\ v \leftarrow \text{LocalUpdate}(\text{PP, ac}_k, op_k, aux') \end{array} \right] = 1$$

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

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.

2.7 Splittable Signatures
A splittable signature scheme for a message space $M$ is a signature scheme whose keys are constrainable to certain subsets of $M$ – 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.

$$\text{Setup}(1^\lambda) \rightarrow \text{sk}_M, \text{vk}_M$$

Setup takes the security parameter $\lambda$ in unary, and outputs a secret key $\text{sk}_M$ and a verification key $\text{vk}_M$ for the whole message space. We will sometimes write the unconstrained keys $\text{sk}_M$ and $\text{vk}_M$ as just $\text{sk}$ and $\text{vk}$, respectively.
Split(\text{sk}_M, m) \rightarrow \text{sk}_{\{m\}}, \text{sk}_{M \setminus \{m\}}, \text{vk}_\varnothing, \text{vk}_{\{m\}}, \text{vk}_{M \setminus \{m\}}

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

\text{Sign}(\text{sk}_S, m) \rightarrow \sigma

\text{Sign} takes a possibly constrained secret key \text{sk}_S and a message \text{m} \in S, and outputs a signature \sigma.

\text{Verify}(\text{vk}, m, \sigma) \rightarrow 0 \text{ or } 1

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

A splittable signature scheme must satisfy the following properties.

**Correctness**

For any message \text{m}*, sample \text{sk}_{\{m^*\}}, \text{sk}_{M \setminus \{m^*\}}, \text{sk}_M, \text{vk}_\varnothing, \text{vk}_{\{m^*\}}, \text{vk}_{M \setminus \{m^*\}}, \text{and} \text{vk}_M \text{ as}

\[(\text{sk}_M, \text{vk}_M) \leftarrow \text{Setup}(1^\lambda)\]

and

\[(\text{sk}_{\{m^*\}}, \text{sk}_{M \setminus \{m^*\}}, \text{vk}_\varnothing, \text{vk}_{\{m^*\}}, \text{vk}_{M \setminus \{m^*\}}) \leftarrow \text{Split}((\text{sk}_M, \text{vk}_M))\]

Correctness requires that with probability 1 over the above sampling:

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

**Verification Key Indistinguishability**

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

Verification Key Indistinguishability requires that the following indistinguishabilities hold:

1. \text{vk}_\varnothing \approx \text{vk}_M
2. \text{sk}_{\{m^*\}}, \text{vk}_{\{m^*\}} \approx \text{sk}_{\{m^*\}}, \text{vk}_M
3. \text{sk}_M \setminus \{m^*\}, \text{vk}_M \setminus \{m^*\} \approx \text{sk}_M \setminus \{m^*\}, \text{vk}_M

**Splitting Indistinguishability**

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

Splitting indistinguishability requires that

\[\text{sk}_{\{m^*\}}, \text{sk}_{M \setminus \{m^*\}}, \text{vk}_{\{m^*\}}, \text{vk}_M \setminus \{m^*\} \approx \text{sk}'_{\{m^*\}}, \text{sk}'_{M \setminus \{m^*\}}, \text{vk}'_{\{m^*\}}, \text{vk}'_M \setminus \{m^*\}\]
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 \( \text{sk}_{\{m\}} \) is just defined as \( \text{Sign}(\text{sk}_M, m) \), and is thus not an output of \( \text{Split} \).
- They generate \( \text{vk}_b \) as an output of \( \text{Setup} \) instead of as an output of \( \text{Split} \).
- They have a separate algorithm \( \text{Sign} \) and \( \text{Sign}_{abo} \) for the different types of signing keys.

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

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 \((\text{KeyGen}, \text{GbPrg}, \text{GbMem})\) is said to be fixed-transcript secure if for all RAM machines \( M_0 \) and \( M_1 \) and all memory configurations \( s \) such that:

- \( |M_0| = |M_1| \)
- \( \mathcal{T}(M_0, s) = \mathcal{T}(M_1, s) \)

for all p.p.t. algorithms \( A \),

\[
\Pr \left[ A(1^\lambda, \tilde{M}_b, \tilde{s}) = b \mid \text{SK} \leftarrow \text{KeyGen}(1^\lambda, S, T), \ b \leftarrow \{0, 1\}, \ \tilde{M}_b \leftarrow \text{GbPrg}(\text{SK}, M_b), \ \tilde{s} \leftarrow \text{GbMem}(\text{SK}, s) \right] \leq \frac{1}{2} + \text{negl}(\lambda)
\]

Theorem 3.1. There exists a fixed-transcript secure garbling scheme for RAM machines.

3.1 Construction

Similarly to [KLW15], the idea here is to keep a Merkle-tree-hash of the memory, using an IO-friendly hash function (namely, the positional accumulator). The root of the tree is signed by the RAM machine, using an IO-friendly signature scheme (namely, the splittable signature). In addition, to help argue global consistency of the computation over multiple steps, the RAM machine uses the iterates to keep a succinct record of its local state.

More precisely let \( M = (\Sigma, Q, Y, C) \) be a RAM machine. Recall that the transition function \( C \) has two inputs – an internal state \( q \in Q \) and a memory symbol \( \sigma \in \Sigma \) – and produces either an “official” output \( y \in Y \), or a tuple \((q', \text{op}) \in Q \times (\mathbb{N} \times \Sigma)\).

Construction 3.2. We define \((\text{KeyGen}, \text{GbPrg}, \text{GbMem})\), and we also describe a RAM algorithm \( \text{Eval} \) in pseudocode for clarity of exposition.

- \( \text{KeyGen}(1^\lambda, T, S) \) samples \((\text{Acc}.\text{PP}, \text{ac}_0, \text{store}_0) \leftarrow \text{SetupAcc}(1^\lambda), \ (\text{itr}.\text{PP}, \text{itr}_0) \leftarrow \text{itr}.\text{Setup}(1^\lambda)\), and a puncturable PRF \( F \leftarrow \text{PPRFF}.\text{KeyGen}(1^\lambda) \). It then outputs \( \text{SK} = (\text{Acc}.\text{PP}, \text{itr}.\text{PP}, \text{ac}_0, \text{store}_0, \text{itr}_0, F) \).
- \( \text{GbMem}(\text{SK}, s) \) computes \((\text{sk}_0^\lambda, \text{vk}_0^\lambda) \leftarrow \text{SetupSpl}(1^\lambda; F(0))\). Let \( a_1 < \ldots < a_{\|s\|_0} \) be the non-empty addresses of \( s \). \( \text{GbMem} \) iteratively computes \( \text{store}_{a} = \text{store}_{a_{\|s\|_0}} \) and \( \text{ac}_{a} = \text{ac}_{a_{\|s\|_0}} \):

  For \( i = 1, \ldots, \|s\|_0 \):
  \( (\text{store}_i, \text{aux}_i) \leftarrow \text{Update}(\text{PP}, \text{store}_{i-1}, \text{ReadWrite}(a_i \mapsto s(a_i))) \)
  \( (\sigma_i, \text{ac}_i) \leftarrow \text{LocalUpdate}(\text{PP}, \text{ac}_{i-1}, \text{ReadWrite}(a_i \mapsto s(a_i)), \text{aux}_i) \)
case derived from these values. We show a sequence of indistinguishable hybrid distributions starting with the

\[\text{KLW15},\] so we just give an overview of these hybrids.

\[\text{fact that the size of } \tilde{\sigma}\text{, the splittable signatures, cryptographic iterators, and positional accumulators. Efficiency follows from the}\]

Proof. Correctness, efficiency, and succinctness are easy to see. Correctness follows from the correctness of

\[\text{Spl,}\] \[\text{Itr,}\] \[\text{Signature}\]

\[\text{If}\]

\[\text{Eval}\]

\[\text{It suffices to show indistinguishability of } (\tilde{\sigma}, \text{state, } \tilde{\sigma}, \text{signature, } \text{auxiliary information})\]

\[\text{An efficient implementation of } \text{Eval} \text{ will reuse the same local variable to store } \text{op}_0, \text{op}_1, \ldots, \text{and similarly with } \text{aux}_1, \text{aux}_2, \ldots \text{ and } \text{ac}_0, \text{ac}_1, \ldots \text{. It should also update } \text{store} \text{ in-place, rather than reading all of } \text{store}_i, \text{ and writing back all of } \text{store}_{i+1}\.\]

\[\text{Input: } \text{Timestamp } t, \text{ state } q, \text{ memory operation } \text{op}, \text{ accumulator } \text{ac}, \text{ iterator } \text{itr}, \text{ signature } \sigma, \text{ auxiliary information } \text{aux}\]

\[\text{Data: } \text{Transition function } C_0, \text{ Puncturable PRF } F, \text{ accumulator public parameters } \text{Acc.PP}, \text{ iterator public parameters } \text{itr.PP}\]

1. \[\text{if } t > T \text{ or } t < 0 \text{ then return } \bot;\]
2. \[\text{(vk}_1^A, \text{sk}_1^A) \leftarrow \text{Spl.Setup}(1^\lambda; F(t));\]
3. \[\text{if } \text{Spl.Verify} (\text{vk}_1^A, (q, \text{op}, \text{ac}, \text{itr}), \sigma) = 0 \text{ then return } \bot;\]
4. \[\text{Parse } \text{Acc.LocalUpdate} (\text{Acc.PP}, \text{ac}, \text{op}, \text{aux}) \text{ as } (\text{y,ac'}) \text{ or else return } \bot;\]
5. \[\text{if } C_0(q, v) = y \in Y \text{ then return } y;\]
6. \[\text{Parse } C_0(q, v) \text{ as } (\text{q', op'});\]
7. \[\text{(vk}_{t+1}^A, \text{sk}_{t+1}^A) \leftarrow \text{Spl.Setup}(1^\lambda; F(t + 1));\]
8. \[\text{itr'} \leftarrow \text{itr.Iterate} (\text{itr.PP}, \text{itr}, (q, \text{op}, \text{ac}, \text{itr}, \text{aux}));\]
9. \[\text{\sigma'} \leftarrow \text{Spl.Sign} (\text{sk}_{t+1}^A, (\text{q', op'}, \text{ac'}, \text{itr}'));\]
10. \[\text{return } (\text{q', op'}, \text{ac'}, \text{itr'}, \sigma');\]

Algorithm 1: Circuit \(P\)

3.2 Proof of Security

Theorem 3.3. If \text{Spl} is a splittable signature scheme, \text{Itr} is a cryptographic iterator, \text{Acc} is a positional accumulator, and \text{iO} is an indistinguishability obfuscator, then Construction 3.2 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 \(C\) depends only polylogarithmically on the time bound \(T\). Succinctness follows from the fact that \text{store} will have \(\tilde{O}(|x|)\) non-empty addresses, and is represented succinctly.

It suffices to show indistinguishability of \((\tilde{P}, \tilde{s}, \text{Acc.PP}, \text{itr.PP})\) in the two cases, as \(M, \tilde{s}\) can be publicly derived from these values. We show a sequence of indistinguishable hybrid distributions starting with the case \(M = M_0\), and ending with the case \(M = M_1\). These hybrids are essentially identical to the ones in [KLIW15], so we just give an overview of these hybrids.

\[\text{Here we are apparently assuming that } M \text{ has a starting state of } \bot \text{ and first reads (and writes) the 0th address, but we can ensure these properties without loss of generality.}\]
Hybrids Overview  At a high level, we only modify the circuit we use the variables in the definition of Eval instead of checking whether the signature \( \sigma \) on \( \langle q, o_p, a_c, i_t, r \rangle \) verifies under one of \( \mathbf{vk}^A_t \) (which is punctured at \( \mathcal{M} \setminus \{ \langle q_0, o_p, a_c, i_t \rangle \} \)) and \( \mathbf{vk}^B_t \) (which is punctured at \( \{ \langle q_0, o_p, a_c, i_t \rangle \} \)), we only check that it verifies under the unpunctured \( \mathbf{vk}^A_t \). This is indistinguishable by the splittable signature’s splitting indistinguishability property.

1. We add a new branch to \( C \). Instead of just checking that \( \mathbf{vk}^A_t \) accepts \( \langle q, o_p, a_c, i_t, \sigma \rangle \), we also check (when \( t \leq T \)) against the key \( \mathbf{vk}^B_t \), which is derived from a different puncturable PRF \( G \). When only \( \mathbf{vk}^B_t \) accepts \( \langle q, o_p, a_c, i_t, \sigma \rangle \), we proceed as before except that we compute with \( C_1 \) instead of \( C_0 \), and we sign outputs with \( \mathbf{sk}^B_{t+1} \) instead of with \( \mathbf{sk}^A_{t+1} \).

The indistinguishability of this change follows by \( O(T) \) applications of the indistinguishability of punctured keys, together with the security of \( iO \).

2. We hard-code \( \mathbf{vk}^A_0 \) and \( \mathbf{vk}^B_0 \), and puncture \( F \) and \( G \) at \( \{0\} \). This change preserves functionality and is hence indistinguishable by \( iO \).

3. We replace \( \mathbf{vk}^A_0 \) and \( \mathbf{vk}^B_0 \) by keys punctured on the sets \( \mathcal{M} \setminus \{ \langle q_0, o_p, a_c, i_t \rangle \} \) and \( \{ \langle q_0, o_p, a_c, i_t \rangle \} \) respectively. These changes are indistinguishable by the indistinguishability of punctured keys.

4. We generate \( \mathbf{Acc} \).PP using \( \mathbf{SetupAccEnforceUpdate} \) so that \( \mathbf{aux}_0 \) is the only value for \( \mathbf{aux} \) such that \( \mathbf{LocalUpdate}(\mathbf{Acc}.PP, a_c, o_p, \mathbf{aux}) \neq ⊥ \). This is in fact equal to \( s_0, a_c \). 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 \( iO \).

6. We generate \( \mathbf{Acc} \).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 \( \mathbf{sk}^A_t \) in branch ‘A’ and \( \mathbf{sk}^B_t \) in branch B, we do the same thing in both branches. Namely, we use \( \mathbf{sk}^A_t \) if and only if \( \langle q, o_p, a_c, i_t \rangle = \langle q_0, o_p, a_c, i_t \rangle \). This is functionally equivalent because \( \mathbf{vk}^A_0 \) and \( \mathbf{vk}^B_0 \) accept disjoint sets of messages, and hence this change is indistinguishable by \( iO \). Note the ‘A’ branch and ‘B’ branch are now identical.

8. We generate \( \mathbf{itr}.PP \) using \( \mathbf{SetupEnforce} \) so that \( \mathbf{itr} = \mathbf{itr}_1 \) if and only if \( \langle q, o_p, a_c, i_t, \mathbf{aux} \rangle \) is equal to \( \langle q_0, o_p, a_c, i_t \rangle \). This change is indistinguishable by the iterator’s setup indistinguishability.

9. Instead of choosing whether to use \( \mathbf{sk}^A_t \) or \( \mathbf{sk}^B_t \) based on the value of \( \langle q, o_p, a_c, i_t \rangle \), we choose based on the value of \( \langle q', o'_p, a'_c, i_t' \rangle \). This is functionally equivalent because \( i_t' = i_t \). This change is indistinguishable by the iterator’s setup indistinguishability.

10. We generate \( \mathbf{itr}.PP \) normally, which is indistinguishable by the iterator’s indistinguishability of setup.

11. Instead of checking whether the signature \( \sigma \) on \( \langle q, a_c, i_t \rangle \) verifies under one of \( \mathbf{vk}^A_0 \) (which is punctured at \( \mathcal{M} \setminus \{ \langle q_0, o_p, a_c, i_t \rangle \} \)) and \( \mathbf{vk}^B_0 \) (which is punctured at \( \{ \langle q_0, o_p, a_c, i_t \rangle \} \)), we only check that it verifies under the unpunctured \( \mathbf{vk}^A_0 \). This is indistinguishable by the splittable signature’s splitting indistinguishability property.

12. We unpuncture \( F \) and \( G \) at 0 and un-hardcode \( \mathbf{vk}^A_0 \) and \( \mathbf{vk}^B_0 \). This is functionally equivalent and hence indistinguishable by \( iO \).

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 \((\text{KeyGen}, \text{GbPrg}, \text{GbMem})\) is said to be fixed-memory secure if for all RAM machines \(M_0\) and \(M_1\), memory configurations \(s\), all time bounds \(T\) and space bounds \(S\) satisfying:

- \(\text{Time}(M_0, s) \leq T\)
- \(\text{Space}(M_0, s) \leq S\)
- \(M_0(s) = M_1(s)\)
- \(|M_0| = |M_1|\)

Writing \(T(M_0, s) = ((q_0, a_0, v_0), \ldots, (q_t, a_t, v_t))\) and \(T(M_1, s) = ((q'_0, a'_0, v'_0), \ldots, (q'_t, a'_t, v'_t))\), it holds that \(t = t'\) and for each \(i \in [t]\), \(a_i = a'_i\) and \(v_i = v'_i\).

it holds that for all p.p.t. adversaries \(A\)

\[
\Pr \left[ A(1^\lambda, M_b, s) = b \begin{array}{l}
SK \leftarrow \text{KeyGen}(1^\lambda, T, S) \\
b \leftarrow \{0, 1\} \\
M_b \leftarrow \text{GbPrg}(SK, M) \\
s \leftarrow \text{GbMem}(SK, s)
\end{array} \right] \leq \frac{1}{2} + \text{negl}(\lambda)
\]

4.1 Construction

Given a garbling scheme \((\text{KeyGen}', \text{GbPrg}', \text{GbMem}')\) satisfying fixed transcript security, we build a garbling scheme \((\text{KeyGen}, \text{GbPrg}, \text{GbMem})\) 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 \((\text{KeyGen}, \text{GbPrg}, \text{GbMem})\):

- \(\text{KeyGen}(1^\lambda, T, S)\) samples \(K \leftarrow \text{KeyGen}'(1^\lambda, T, S)\) and a puncturable PRF \(F\), and outputs \((K, F, T, S)\).
- \(\text{GbPrg}((K, F, T, S), M = (\Sigma, Q, Y, C))\) outputs \(\text{GbPrg}'(K, M' = (\Sigma, Q', Y, C'))\), where \(Q' = [T] \times \{0, 1\}^{|Q|}\), and \(C'\) is defined in Algorithm 2. \(q_0\), the initial state for \(M'\), is defined as \((0, F(0) \oplus q_0)\).
- \(\text{GbMem}((K, F, T, S), s)\) outputs \(\text{GbMem}(K, s)\).

```
Input: state \(q\), memory symbol \(\sigma\)
Data: Puncturable PRF \(F\), underlying transition function \(C\)
1 Parse \(q\) as \((t, c_q)\);
2 \(q_{\text{in}} := F(t) \oplus c_q;\)
3 \(\text{out} := C(q_{\text{in}}, \sigma);\)
4 if \(\text{out} \in Y\) then return \(\text{out}\);
5 else
6 | Parse \(q_{\text{out}}\) as \((q_{\text{out}}, \text{op});\)
7 | return \(((t + 1, F(t + 1) \oplus q_{\text{out}}), \text{op});\)
8 end
```

Algorithm 2: Transition function \(C'\)
4.2 Proof of Security

Theorem 4.2. If $(\text{KeyGen}', \text{GbPrg}', \text{GbMem}')$ 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. Given RAM machines $M_0$ and $M_1$ and a memory configuration $s$, a time bound $T$, and a space bound $S$, recall we want to show the indistinguishability of two “real-world” distributions, which we denote $RW_0$ and $RW_1$. Let $t^*$ denote $\text{Time}(M,v)$. We first define $t^* + 1$ hybrid distributions $H_0, \ldots, H_{t^*}$.

Real World $b$: For $b \in \{0, 1\}$, the distribution $RW_b$ is defined as the distribution on $(\tilde{M}, \tilde{s})$ obtained by sampling:

1. $K_{FT} \leftarrow \text{KeyGen}'(1^\lambda, T, S)$, and a puncturable PRF $F$.
2. $\tilde{M} \leftarrow \text{GbPrg}'(K_{FT}, M_b')$, where the RAM machine $M_b' = (\Sigma, Q', Y, C_b')$ with $Q' = [T] \times Q$. The transition function $C_b'$ is given in Algorithm 2, with the hard-coded $C$ equal to the transition function of $M_b$.
3. $\tilde{s} \leftarrow \text{GbMem}'(K_{FT}, s)$.

Hybrid $H_i$: Hybrid $H_i$ is defined as the distribution on $(\tilde{M}, \tilde{s})$ obtained by sampling:

1. $K_{FT} \leftarrow \text{KeyGen}'(1^\lambda, T, S)$, and a puncturable PRF $F$.
2. $\tilde{M} \leftarrow \text{GbPrg}'(K_{FT}, M_i')$, where the RAM machine $M_i' = (\Sigma, Q', Y, C_i')$ with $Q' = [T] \times Q$. The transition function $C_i'$ is given in Algorithm 3, with the hard-coded $C_0$ equal to the transition function of $M_0$ and $C_1$ equal to the transition function of $M_1$. If the $t_x - i$th internal state (respectively memory operation) in $T(M_i, s)$ is $q_{t_x-i}$ (respectively $\text{op}_{t_x-i}$), then the hard-coded constant $q_{t_x-i}$ in $C_i'$ is equal to $(t_x - i, F(t_x - i) \oplus q_{t_x-i})$, and $\text{op}_{t_x-i} = \text{op}_{t_x-i}$.
3. $\tilde{s} \leftarrow \text{GbMem}'(K_{FT}, s)$.

\begin{algorithm}[h]
\caption{Transition function $C_{0,i}$}
\begin{algorithmic}[1]
\State \textbf{Input:} state $q$, memory symbol $\sigma$
\State \textbf{Data:} Puncturable PRF $F$, underlying transition functions $C_0$ and $C_1$, a string $q_{t_x-i}$, and operation $\text{op}_{t_x-i}$
\State \textbf{1} Parse $q$ as $(t, c_q)$;
\State \textbf{2} if $t = t_x - i$ then \textbf{return} $(q^*, \text{op}^*)$;
\State \textbf{3} if $t < t_x - i$ then $(q_{\text{out}}, \text{op}) \leftarrow C_0(F(t) \oplus c_q, \sigma)$;
\State \textbf{4} else $(q_{\text{out}}, \text{op}) \leftarrow C_1(F(t) \oplus c_q, \sigma)$;
\State \textbf{5} return $((t+1, F(t+1) \oplus q_{\text{out}}), \text{op})$;
\end{algorithmic}
\end{algorithm}

Evidently $H_0$ is indistinguishable from $RW_0$ and $H_{t^*}$ is indistinguishable from $RW_1$ by the security of the fixed transcript garbler. In order to complete the proof that $RW_0 \approx RW_1$, 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 define 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 $H_{i, 1}$: Hybrid $H_{i, 1}$ is defined as the distribution on $(\tilde{M}, \tilde{s})$ obtained by sampling:

1. $K_{FT} \leftarrow \text{KeyGen}(1^\lambda, T, S)$, and a puncturable PRF $F$.

2. $\tilde{M} \leftarrow \text{GbPrg}(K_{FT}, M_{1, i})$, where the RAM machine $M_{1, i} = (\Sigma, Q', Y, C_{1, i}')$ with $Q' = [T] \times Q$. The transition function $C_{1, i}'$ is given in Algorithm 4, with the hard-coded $C_0$ equal to the transition function of $M_0$, and $C_1$ equal to the transition function of $M_1$.

   If the $t - i$th internal state (respectively memory operation) in $\mathcal{T}(M_1, s)$ is $q_{t, i}^1$ (respectively $\text{op}_{t, i}^1$), then the hard-coded constant $q_{t, i - 1}$ in $C_{1, i}'$ is equal to $(t - i, F(t - i) \oplus q_{t, i - 1})$, and $\text{op}_{t, i - 1} = \text{op}_{t, i}$.

   If the $t - i - 1$th internal state (respectively memory operation) in $\mathcal{T}(M_0, s)$ is $q_{t, i - 1}^0$ (respectively $\text{op}_{t, i - 1}^0$), then the hard-coded constant $q_{t, i - 1}$ in $C_{1, i}'$ is equal to $(t - i - 1, F(t - i - 1) \oplus q_{t, i - 1}^0)$, and $\text{op}_{t, i - 1} = \text{op}_{t, i - 1}^0$.

3. $\tilde{s} \leftarrow \text{GbMem}(K_{FT}, s)$.

```markdown
Input: state $q$, memory symbol $\sigma$
Data: Punctured PRF $F' = F\{t - i\}$, underlying transition functions $C_0$ and $C_1$, strings $q_{t, i - 1}$ and $q_{t, i}$, operations $\text{op}_{t, i - 1}$ and $\text{op}_{t, i}$, output $y = M_0(s)$

1. Parse $q$ as $(t, c_q)$;
2. if $t = t_i$ then return $y$;
3. if $t = t_i - i$ then return $(q_{t, i - 1}, \text{op}_{t, i - 1})$;
4. if $t = t_i - i$ then return $(q_{t, i - 1}, \text{op}_{t, i - 1})$;
5. else if $t < t_i - i$ then $(q_{\text{out}}, \text{op}) \leftarrow C_0(F(t) \oplus c_q, \sigma)$;
6. else $(q_{\text{out}}, \text{op}) \leftarrow C_1(F(t) \oplus c_q, \sigma)$;
7. return $((t + 1, F(t + 1) \oplus q_{\text{out}}), \text{op})$
```

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

Hybrid $H_{i, 2}$: Hybrid $H_{i, 2}$ is defined as the distribution on $(\tilde{M}, \tilde{s})$ obtained by sampling:

1. $K_{FT} \leftarrow \text{KeyGen}(1^\lambda, T, S)$, and a puncturable PRF $F$.

2. $\tilde{M} \leftarrow \text{GbPrg}(K_{FT}, M_{1, i})$, where the RAM machine $M_{1, i} = (\Sigma, Q', Y, C_{1, i}'')$ with $Q' = [T] \times Q$. The transition function $C_{1, i}''$ is given in Algorithm 4, with the hard-coded $C_0$ equal to the transition function of $M_0$, and $C_1$ equal to the transition function of $M_1$.

   If the $t - i$th internal state (respectively memory operation) in $\mathcal{T}(M_1, s)$ is $q_{t, i}^1$ (respectively $\text{op}_{t, i}^1$), then the hard-coded constant $q_{t, i - 1}$ in $C_{1, i}''$ is equal to $(t - i, F(t - i) \oplus q_{t, i - 1})$, and $\text{op}_{t, i - 1} = \text{op}_{t, i}$.

   $q_{t, i - 1}$ and $\text{op}_{t, i - 1}$ are defined analogously.

3. $\tilde{s} \leftarrow \text{GbMem}(K_{FT}, s)$.

We now need to show that

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

The first and third indistinguishabilities are shown via reduction to the fixed transcript security of the underlying garbling scheme. The second indistinguishability is shown via reduction to the pseudorandomness of $F$ at the selectively punctured point $t - i$.

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 \((\text{KeyGen}, \text{GbPrg}, \text{GbMem})\) is said to be fixed-address secure if for all RAM machines \(M\), memory configurations \(s\), time bounds \(T\), and space bounds \(S\) satisfying the following conditions:

- \(\text{Space}(M_0, s_0) \leq S\) and \(\text{Time}(M_0, s_0) \leq T\).
- \(\{a : s_0(a) \neq \epsilon\} = \{a : s_1(a) \neq \epsilon\}\)
- If \(T(M_0, s) = ((q_0, a_0, v_0), \ldots, (q_{t-1}, a_{t-1}, v_{t-1}))\) and \(T(M_1, s) = ((q'_0, a'_0, v'_0), \ldots, (q'_{t'-1}, a'_{t'-1}, v'_{t'-1}))\), then \(t = t'\) and for each \(i \in \llbracket t \rrbracket\), \(a_i = a'_i\).
- \(|M_0| = |M_1|\)

for all p.p.t. adversaries \(A\), it holds that

\[
\Pr \left[ A(1^\lambda, M_0, s) = b \right] = \frac{1}{2} + \text{negl}(\lambda)
\]

5.1 Construction

Given a garbling scheme \((\text{KeyGen}', \text{GbPrg}', \text{GbMem}')\) satisfying fixed-memory security, we build a garbling scheme \((\text{KeyGen}, \text{GbPrg}, \text{GbMem})\) satisfying fixed-address security.

Overview. Our construction of \(\text{Garble}(M, x, T, S)\) applies \(\text{Garble}'\) 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’ and ‘B’ tracks by modifying the memory alphabet \(\Sigma\) to hold two symbols.
- \(M'\) writes metadata alongside each value to indicate the time and address at which it is written.
- \(M'\) authenticates each value it writes: instead of writing \((t, a, v)\) to an address \(a\), it writes \((t, a, F((t, a)) \oplus v, G((t, a)) \oplus v, w, F'((t, a)) \oplus v, G'((t, a)) \oplus v, \ldots)\), where \(F\) and \(G\) are puncturable pseudorandom functions.

Construction 5.1. We define \((\text{KeyGen}, \text{GbPrg}, \text{GbMem})\):

- \(\text{KeyGen}(1^\lambda, T, S)\) samples \(K \leftarrow \text{KeyGen}'(1^\lambda, T, S)\), as well as puncturable PRFs \(F\) and \(G\) and outputs \((K, F, G, T, S)\).
- \(\text{GbPrg}((K, F, G, T, S), M = (\Sigma, Q, Y, C))\) outputs \(\text{GbPrg}'(K, M' = (\Sigma', Q', Y, C'))\), where \(\Sigma' = [T] \times \llbracket S \rrbracket \times \llbracket 0, 1 \rrbracket^{2\lambda} \times \llbracket 0, 1 \rrbracket^{2\lambda}, Q' = [T] \times Q \times Q\), and \(C'\) is defined as Algorithm 5. The initial state \(q'_0 \in Q'\) of \(M'\) is defined as \((0, q_0, q_0)\).
• \( \text{GbMem}(K, F, G, T, S, s) \) samples \( \tilde{s} \leftarrow \text{GbMem}(K, s') \), where

\[
s'(a) = \begin{cases} 
(0, a, F((0, a)) \oplus s(a), G((0, a)) \oplus s(a)) & \text{if } s(a) \neq \epsilon \\
\epsilon & \text{otherwise}
\end{cases}
\]

**Data:** Underlying transition function \( C \), puncturable PRFs \( F \) and \( G \\
**Input:** State \( q \), symbol \( \sigma \)

1. Parse \( q \) as \((t_q, q_A, q_B)\);
2. Parse \( \sigma \) as \((t_\sigma, a_\sigma, \sigma_A, \sigma_B)\);
3. Compute \( \text{out} \leftarrow C(q_A, F((t_\sigma, a_\sigma)) \oplus \sigma_A)\);
4. If \( \text{out} \in Y \) then return \( \text{out} \);
5. Parse \( \text{out} \) as \((q_{\text{out}}, (a_{\text{out}}, \text{op}_{\text{out}}, \sigma_{\text{out}}))\);
6. Return \((t_q + 1, q_{\text{out}}, q_{\text{out}}, (a_{\text{out}}, \text{op}_{\text{out}}, F((t_q, a_{\text{out}})) \oplus \sigma_{\text{out}}, G((t_q, a_{\text{out}})) \oplus \sigma_{\text{out}}))\);

**Algorithm 5:** \( C' \)

### 5.2 Proof of Security

**Theorem 5.2.** If \((\text{KeyGen}', \text{GbPrg}', \text{GbMem}')\) is a 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.** For RAM machines \( M_0 \) and \( M_1 \), memory configurations \( s_0 \) and \( s_1 \), and and a time bound \( T \) and a space bound \( S \), we want to show that two “real world” distributions, which we will denote by \( RW_0 \) and \( RW_1 \), are indistinguishable.

**Real world \( b \):** The distribution \( RW_b \) is defined as the distribution on \((\tilde{M}, \tilde{s})\) obtained by sampling:

1. \( K_{FM} \leftarrow \text{KeyGen}'(1^\lambda, T, S) \), and puncturable PRFs \( F \) and \( G \).
2. \( \tilde{M} \leftarrow \text{GbPrg}'(K_{FM}, M'_b) \), where the RAM machine \( M'_b \)’s transition function \( C'_b \) is given in Algorithm 5.
   In \( C'_b \), the hard-coded transition function \( C \) is given by the transition function of \( M_b \).
3. \( \tilde{s} \leftarrow \text{GbMem}(K_{FM}, s'_b) \), where

\[
s'_b(a) = \begin{cases} 
(0, a, F((0, a)) \oplus s_b(a), G((0, a)) \oplus s_b(a)) & \text{if } s_b(a) \neq \epsilon \\
\epsilon & \text{otherwise}
\end{cases}
\]

**Hybrid \( H_{01} \):** Hybrid \( H_{01} \) is defined as the distribution on \((\tilde{M}, \tilde{s})\) obtained by sampling:

1. \( K_{FM} \leftarrow \text{KeyGen}'(1^\lambda, T, S) \), and puncturable PRFs \( F \) and \( G \).
2. \( \tilde{M} \leftarrow \text{GbPrg}'(K_{FM}, M'_{01}) \), where the RAM machine \( M'_{01} \)’s transition function \( C'_{01} \) is given in Algorithm 6.
3. \( \tilde{s} \leftarrow \text{GbMem}(K_{FM}, s'_{01}) \), where

\[
s'_{01}(a) = \begin{cases} 
(0, a, F((0, a)) \oplus s_0(a), G((0, a)) \oplus s_1(a)) & \text{if } s_0(a) \neq \epsilon \\
\epsilon & \text{otherwise}
\end{cases}
\]

**Lemma 5.3.** \( RW_0 \approx H_{01} \)

**Proof.** (of Lemma 5.3)

We show a sequence of \( t^* + 1 \) indistinguishable hybrid distributions \( H_{00,i} \) for \( i = 0, \ldots, t^* \) such that

\[ H_0 \approx H_{00,0} \approx \cdots \approx H_{00,t^*} \approx H_{01}. \]
Data: Transition functions $C_0$ and $C_1$, Puncturable PRF $F$
Input: State $q$, symbol $\sigma$
1. Parse $q$ as $(t_q, q_A, q_B)$;
2. Parse $\sigma$ as $(t_\sigma, a_\sigma, \sigma_A, \sigma_B)$;
3. Compute $out_A \leftarrow C_0(q_A, F((t_\sigma, a_\sigma)) \oplus \sigma_A)$;
4. Compute $out_B \leftarrow C_1(q_B, G((t_\sigma, a_\sigma)) \oplus \sigma_B)$;
5. if $out_A \in Y$ then return $out_A$;
6. Parse $out_A$ as $(a^{out}_{A}, (\sigma^{out}_{A}))$;
7. Parse $out_B$ as $(a^{out}_{B}, (\sigma^{out}_{B}))$ // In honest execution, $a^{out}_{A} = a^{out}_{B}$
8. return $((t_q + 1, q^{out}, q^{out}), (a^{out}_{A}, (t_q, a^{out}_{A}) \oplus \sigma^{out}_{A}), G((t_q, a^{out}_{A})) \oplus \sigma^{out}_{B})$;

Algorithm 6: RAM machine $M_{01}$

Hybrid $H_{00,i}$: Informally, $H_{00,i}$ is the garbling of a machine $M'_{00,i}$ and memory $s'_{00,i}$ which execute $M_0$ 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, Hybrid $H_{00,i}$ is defined as the distribution on $(\tilde{M}, \tilde{s})$ obtained by sampling:
1. $K_{FM} \leftarrow \text{KeyGen}'(1^\lambda, T, S)$, and puncturable PRFs $F$ and $G$.
2. $\tilde{M} \leftarrow \text{GbPr}((K_{FM}, M'_{00,i}))$, where the RAM machine $M'_{00,i}$’s transition function $C'_{00,i}$ is given in Algorithm 7.
3. $\tilde{s} \leftarrow \text{GbMem}(K_{FM}, s'_{00,i})$, where

$$s'_{00,i}(a) = \begin{cases} (0, a, F((0, a)) \oplus s_0(a), G((0, a)) \oplus s_1(a)) & \text{if } s_0(a) \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

Data: RAM transition functions $C_0$ and $C_1$, Puncturable PRF $F$ and $G$
Input: State $q$, symbol $\sigma$
1. Parse $q$ as $(t_q, q_A, q_B)$;
2. Parse $\sigma$ as $(t_\sigma, a_\sigma, \sigma_A, \sigma_B)$;
3. Compute $out_A \leftarrow C_0(q_A, F((t_\sigma, a_\sigma)) \oplus \sigma_A)$;
4. Compute $out_B \leftarrow C_1(q_B, G((t_\sigma, a_\sigma)) \oplus \sigma_B)$;
5. if $out_A \in Y$ then return $out_A$;
6. if $t_q \leq i$ then
7. Compute $out_B \leftarrow C_1(q_B, G((t_\sigma, a_\sigma)) \oplus \sigma_B)$;
8. Parse $out_B$ as $(a^{out}_{B}, (\sigma^{out}_{B}))$ // In honest execution, $a^{out}_{A} = a^{out}_{B}$
9. return $((t_q + 1, q^{out}, q^{out}), (a^{out}_{A}, (t_q, a^{out}_{A}) \oplus \sigma^{out}_{A}), G((t_q, a^{out}_{A})) \oplus \sigma^{out}_{B})$;
else return $((t_q + 1, q^{out}, q^{out}), (a^{out}_{A}, (t_q, a^{out}_{A}) \oplus \sigma^{out}_{A}), G((t_q, a^{out}_{A})) \oplus \sigma^{out}_{A})$;

Algorithm 7: Algorithm $M_{00,i}$

**Claim 5.3.1.** $RW_0 \approx H_{00,0}$

**Proof.** Starting with the process for sampling $RW_0$, one can first replace $M_0$ by $M'_{00,0}$ without changing the addresses accessed. By fixed-address security of the underlying garbling scheme, this is an indistinguishable change.

We now only need to change $s'_0$ into $s'_{00,0}$ – in other words, changing track ‘B’ to initially store encryptions of $s_1$ instead of $s_0$. This is indistinguishable because $M'_{00,0}$ never needs to decrypt (or encrypt) values from
Proof. We make a sequence of indistinguishable changes to the process for sampling $H_{00,i}$, and end up with a process for sampling $M_{00,i}$. In particular, we modify the machine $M'_{00,i}$ until we get $M_{00,i}$.

Let $c_i^0$ denote the value written by $M_0$ at time $i$, and let $c_i^1$ 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 $(i, c_j^0)$ to address $a_j$ on track B, where $c_j^0$ is a hard-coded ciphertext equal to $G((i, a_j)) \oplus c_j^0$. We also change it to only use a punctured PRF $G' = G\{i, a_j\}$, as in Algorithm 8 (with hard-coded $c^\star = c_j^0$). This change is indistinguishable by fixed memory security.

Next the hard-coded value for $c^\star$ is changed from $c_j^0$ to $c_j^1 = (i, G((i, a_j)) \oplus c_j^1)$. The indistinguishability of this change follows from the pseudorandomness of $G$ at the selectively punctured point $(i, a_j)$.

After these changes, $M'_{00,i-1}$ on $s'_{00,i-1}$ accesses the same addresses and writes the same values as $M_{00,i}$ on $s'_{00,i}$. So by the fixed memory security of the underlying garbling scheme, this hybrid is indistinguishable from $H_{00,i}$.

\begin{algorithm}
\caption{Hybrid transition function $C'$}
\begin{algorithmic}
\State \textbf{Data:} RAM transition functions $C_0$ and $C_1$, ciphertext $c^\star$, Puncturable PRF $F$, Punctured PRF $G' = G\{i, a_j\}$
\State \textbf{Input:} State $q$, symbol $\sigma$
\State 1 Parse $q$ as $(t_q, q_A, q_B)$;
\State 2 Parse $\sigma$ as $(t_\sigma, a_\sigma, \sigma_A, \sigma_B)$;
\State 3 Compute $\text{out}_A \leftarrow C_0(q_A, F((t_\sigma, a_\sigma)) \oplus \sigma_A)$;
\State 4 Compute $\text{out}_B \leftarrow C_1(q_B, G((t_\sigma, a_\sigma)) \oplus \sigma_B)$;
\State 5 if $\text{out}_A \in Y$ then return $\text{out}_A$;
\State 6 Parse $\text{out}_A$ as $(\sigma_A, \text{out}_A)$;
\State 7 Parse $\text{out}_B$ as $(\sigma_B, \text{out}_B)$ // In honest execution, $a^{\text{out}}_A = a^{\text{out}}_B$
\State 8 if $t_q < i$ then
\State 9 \quad \quad return $((t_q + 1, q_A, q_B), (a^{\text{out}}_A, (t_q, a^{\text{out}}_A)) \oplus \sigma_A, G((t_q, a^{\text{out}}_A)) \oplus \sigma_B)$;
\State 10 else if $t_q = i$ then
\State 11 \quad \quad return $((t_q + 1, q_A, q_B), (a^{\text{out}}_A, (t_q, a^{\text{out}}_A) + F((t_q, a^{\text{out}}_A)) \oplus \sigma_A, c^\star)$;
\State 12 else return $((t_q + 1, q_A, q_B), (a^{\text{out}}_A, (t_q, a^{\text{out}}_A) + F((t_q, a^{\text{out}}_A)) \oplus \sigma_A, G((t_q, a^{\text{out}}_A)) \oplus \sigma_B)$
\end{algorithmic}
\end{algorithm}

Claim 5.3.3. $H_{00, t^*} \approx H_{01}$

Proof. Line 10 of $C'_{00, t^*}$ (described in Algorithm 7) is never activated on input $s'_{00, t^*}$ because $M_0$ terminates after $t^*$ steps. It’s easy to see then that $M'_{00, t^*}$, on input $s'_{00, t^*}$, accesses the same addresses and writes the same values as $M'_{01}$ on input $s'_{01}$. So the claim follows from fixed-memory security.

Lemma 5.3 follows by combining claims 5.3.1, 5.3.2, and 5.3.3.

Lemma 5.4. $H_{01} \approx RW_1$

Proof. This follows analogously and symmetrically to Lemma 5.3.

The fixed-access security of $(\text{KeyGen}, \text{GbPrf}, \text{GbMem})$ follows from Lemmas 5.3 and 5.4.

\begin{enumerate}
\item \textit{Proof.}
\item \textit{Proof.}
\item \textit{Proof.}
\item \textit{Proof.}
\item \textit{Proof.}
\item \textit{Proof.}
\item \textit{Proof.}
\item \textit{Proof.}
\item \textit{Proof.}
\end{enumerate}
6 Full Security

This section constructs a secure garbling scheme for RAM machines, as in defined in Section 2.4, from any fixed-address garbling scheme. As sketched and motivated in the Introduction, this is done by combining the fixed address garbling scheme with an oblivious RAM (ORAM) scheme that has a special property, namely localized randomness. 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

6.1.1 Syntax

An oblivious RAM is a tuple of p.p.t. algorithms (Setup, OMem, OProg)

- Setup(1^λ, S) takes a security parameter in unary and a space bound S, and outputs a secret key SK.
- OProg(SK, M) takes a secret key SK and a RAM machine M, and outputs a probabilistic RAM machine M’.
- OMem(SK, s) takes a secret key SK and a memory configuration s, and outputs a memory configuration s’

6.1.2 Correctness

For all RAM machines M, space bounds S, and memory configurations s such that \text{Space}(M, s) \leq S,

\[ \Pr \left[ M'(s') = M(s) \middle| \begin{array}{l} SK \leftarrow \text{Setup}(1^\lambda, S) \\ M' \leftarrow \text{OProg}(SK, M) \\ s' \leftarrow \text{OMem}(SK, s) \end{array} \right] \geq 1 - \text{negl}(\lambda) \]

6.1.3 Efficiency

There is a function

\[ \eta : \mathbb{N} \rightarrow \mathbb{N} \]

\[ \eta(S) = \Theta(\text{polylog}(S)) \]

such that whenever \text{Space}(M, s) \leq S for a RAM machine M, a memory configuration s, and a space bound S,

\[ \Pr \left[ \text{Time}(M', s') = \eta(S) \cdot \text{Time}(M, s) \middle| \begin{array}{l} SK \leftarrow \text{Setup}(1^\lambda, S) \\ M' \leftarrow \text{OProg}(SK, M) \\ s' \leftarrow \text{OMem}(SK, s) \end{array} \right] = 1. \]

6.1.4 Localized Randomness

Let \( T = \text{Time}(M, s) \) and \( \eta = \eta(S) \) for some RAM machine M, memory configuration s, and space bound S. Consider the deterministic function

\[ \text{addr}'_{M, s, \lambda} : \{0, 1\}^N \rightarrow \mathbb{N}^{\eta T} \]

\[ \text{addr}'_{M, s, \lambda}(\vec{r}) = \vec{a} \]

where \( \vec{r} \) is used as a random tape to sequentially sample

\[ \begin{array}{l} SK \leftarrow \text{Setup}(1^\lambda, S) \\ M' \leftarrow \text{OProg}(SK, M) \\ s' \leftarrow \text{OMem}(SK, s) \\ \vec{a} \leftarrow \text{addr}(M', s') \end{array} \]
Definition 6.1. An ORAM (Setup, OProg, OMem) is said to have localized randomness if there is a deterministic algorithm Sim such that for all RAM machines $M$, memory configurations $s$, and space bounds $S \geq \text{Space}(M, s)$ and $\text{Time}(M, s) = t$, there exist sets $R_1, \ldots, R_t \subseteq \mathbb{N}$ such that:

- For each $i$, $|R_i| \leq \text{poly}(\log S, \lambda)$.
- For each $i \neq j$, $R_i \cap R_j = \emptyset$.
- For each $i$,
  $$\Pr[\text{addr}'_{M,s,S,\lambda}(\vec{r})_{\eta(i-1),\ldots,\eta(i-1)} = \text{Sim}(\vec{r}_{R_i}) | \vec{r} \leftarrow \{0, 1\}^N] \geq 1 - \text{negl}(\lambda).$$

In Appendix A, we observe that a modification of the Chung-Pass ORAM [CP13] satisfies these properties.

6.2 Construction

Our garbling scheme is very simple; essentially, we just compose the fixed address garbler on top of an ORAM scheme with localized randomness. That is, to garble a machine $M$, we first transform it via the ORAM, and then apply the fixed address garbler to that transformed machine.

Construction 6.1. Let $(\text{KeyGen}', \text{GbPrg}', \text{GbMem}')$ be a fixed-address garbling scheme, and let $(\text{Setup}, \text{OProg}, \text{OMem})$ be an ORAM scheme with localized randomness. We define a garbling scheme $(\text{KeyGen}, \text{GbPrg}, \text{GbMem})$:

- $\text{KeyGen}(1^\lambda, T, S)$ samples $K_{FA} \leftarrow \text{KeyGen}'(1^\lambda, T, S)$ and $K_{ORAM} \leftarrow \text{Setup}(1^\lambda, S)$, and a puncturable PRF $F$ and outputs $(K_{FA}, K_{ORAM}, F)$.
- $\text{GbPrg}((K_{FA}, K_{ORAM}, F), M)$ outputs $\text{GbPrg}'(K_{FA}, \text{OProg}(K_{ORAM}, M)^F)$.
- $\text{GbMem}((K_{FA}, K_{ORAM}, s))$ outputs $\text{GbMem}'(K_{FA}, \text{OMem}(K_{ORAM}, s))$.

6.3 Security Proof

Theorem 6.2. If $(\text{KeyGen}', \text{GbPrg}', \text{GbMem}')$ is a fixed address secure garbling scheme for RAM machines, then Construction 6.1 defines a (fully secure) garbling scheme for RAM machines.

Proof Overview. We proceed through a sequence of hybrid distributions, changing $M'$ so that instead of running OProg($M$), it accesses a simulated sequence of memory addresses. We make this change one timestep at a time, so in hybrid $t^* - i$, OProg($M$) is run for $i$ steps before switching to the simulated access pattern.

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 the memory configuration $\tilde{s}$ to be simulatable given $\|s\|_0$, the number of non-empty addresses of $s$.

Proof. For any RAM machine $M$ and memory configuration $s$ with $\text{Time}(M, s) = t^* \leq T$ and $\text{Space}(M, s) \leq S$, we give a sequence of $t^* + 2$ indistinguishable hybrid distributions $H_0$ through $H_{t^*+1}$.

Real World. We want to show the simulability of the distribution on $(\tilde{M}, \tilde{S})$ which is obtained by sampling:

1. $K_{FA} \leftarrow \text{KeyGen}'(1^\lambda, T, S)$, and $K_{ORAM} \leftarrow \text{Setup}(1^\lambda, S)$.
2. $\tilde{M} \leftarrow \text{GbPrg}(K_{FA}, M)$
3. $\tilde{s} \leftarrow \text{GbMem}(K_{FA}, \text{OMem}(K_{ORAM}, s))$. 

23
Hybrid \(H_i\): For \(0 \leq i \leq t^*\), Hybrid \(H_i\) is defined as the distribution on \((\tilde{M}, \tilde{s})\) obtained by sampling:

1. \(K_{FA} \leftarrow \text{KeyGen}'(1^\lambda, T, S)\), \(K_{ORAM} \leftarrow \text{Setup}(1^\lambda, S)\), and puncturable PRFs \(F\) and \(G\).
2. \(\tilde{M} \leftarrow \text{GbPrg}(K_{FA}, M_i)\), where the RAM machine \(M_i\)'s transition function is given in Algorithm 9.
3. \(\tilde{s} \leftarrow \text{GbMem}(K_{FA}, \text{OMem}(K_{ORAM}, s))\).

**Algorithm 9:** Hybrid transition function \(C_i\).

Hybrid \(H_{t^*+1}\): Hybrid \(H_{t^*+1}\) is defined as the distribution on \((\tilde{M}, \tilde{s})\) obtained by sampling:

1. \(K_{FA} \leftarrow \text{KeyGen}'(1^\lambda, T, S)\), \(K_{ORAM} \leftarrow \text{Setup}(1^\lambda, S)\), and puncturable PRFs \(F\) and \(G\).
2. \(\tilde{M} \leftarrow \text{GbPrg}(K_{FA}, M_{t^*+1})\), where the RAM machine \(M_{t^*+1}\)'s transition function is given in Algorithm 10. \(M_{t^*+1}\)'s initial state is \((0, \bot)\).
3. \(\tilde{s} \leftarrow \text{GbMem}(K_{FA}, \text{OMem}(K_{ORAM}, s_\bot))\), where

\[
s_\bot(a) = \begin{cases} 
\bot & \text{if } a < \|s\|_0 \\
\epsilon & \text{otherwise}
\end{cases}
\]

**Algorithm 10:** Hybrid transition function \(C_{t^*+1}\).

One can easily check that \(H_{t^*+1}\) can be sampled given only \(M(s), \|s\|_0, T, S\), and \(\text{Time}(M, s)\). It remains to prove the following three lemmas.

**Lemma 6.3.** The “real world” distribution is indistinguishable from \(H_0\).

**Lemma 6.4.** For all \(i\) with \(0 \leq i < t^*\), \(H_i \approx H_{i+1}\).

**Lemma 6.5.** \(H_{t^*} \approx H_{t^*+1}\)

Lemmas 6.3 and 6.5 follow immediately from fixed-access security.

**Proof.** (of Lemma 6.4)
We give a sequence of indistinguishable hybrid distributions \( H_i \approx H_{i,1} \approx \cdots \approx H_{i,4} \approx H_{i+1} \).

**Hybrid \( H_{i,1} \):** Hybrid \( H_{i,1} \) is defined as the distribution on \((\tilde{M}, \tilde{s})\) obtained by sampling:

1. \( K_{FA} \leftarrow \text{KeyGen}'(1^\lambda, T, S), \ K_{ORAM} \leftarrow \text{Setup}(1^\lambda, S) \), and puncturable PRFs \( F \) and \( G \).
2. \( \tilde{M} \leftarrow \text{GbPrg}(K_{FA}, M_{i,1}) \), where the RAM machine \( M_{i,1} \)'s transition function is given in Algorithm 11, with the hard-coded values \( p, \iota_1, \ldots, \iota_p \) and \( b_1, \ldots, b_p \) defined so that \( S_i = \{\iota_1, \ldots, \iota_p\} \), and \( b_j = F(\iota_j) \). The hard-coded values \( \vec{a} = (a_0, \ldots, a_{\eta-1}) \) are defined as \( \vec{a} = \text{Sim}(b_1, \ldots, b_p) \).
3. \( \tilde{s} \leftarrow \text{GbMem}(K_{FA}, \text{OMem}(K_{ORAM}, s)) \).

**Data:** Punctured PRF \( F' = F\{S_i\} \), puncturable PRF \( G \), underlying transition function \( C' \),
\[ y = M(s) \], the set \( S_i = \{\iota_1, \ldots, \iota_p\} \), bits \( b_1, \ldots, b_p \), addresses \( a_0, \ldots, a_{\eta-1} \), the running time \( t^* = \text{Time}(M, s) \)

**Input:** State \((t, q_{in})\), symbol \( \sigma \)

1. Let \( (i', j') \) be integers such that \( t = \eta i' + j' \) for \( 0 \leq j' < \eta \);
2. Define \( \bar{F}' \) such that \( \bar{F}'(x) = \begin{cases} b_j & \text{if } x = \iota_j \\ F'(x) & \text{otherwise} \end{cases} \);
3. if \( i' < i \) then
   4. \((q_{out}, \text{op}) := \text{C'}(q_{in}, \sigma)\);
   5. \text{return } ((t+1, q_{out}), \text{op})
4. else if \( i' = i \) then \text{return } ((t+1, q_{in}), (a_j, \bot))
5. else if \( i < i' < t^* \) then \text{return } ((t+1, q_{in}), (\text{Sim}(G(i')_{j'}), \bot))
6. else \text{return } y;

**Algorithm 11:** Hybrid transition function \( C_{i,1} \).

**Hybrid \( H_{i,2} \):** Hybrid \( H_{i,2} \) is defined as the distribution on \((\tilde{M}, \tilde{s})\) obtained by sampling:

1. \( K_{FA} \leftarrow \text{KeyGen}'(1^\lambda, T, S), \ K_{ORAM} \leftarrow \text{Setup}(1^\lambda, S) \), and puncturable PRFs \( F \) and \( G \).
2. \( \tilde{M} \leftarrow \text{GbPrg}(K_{FA}, M_{i,2}) \), where the RAM machine \( M_{i,2} \)'s transition function is given in Algorithm 11, with the hard-coded values \( p, \iota_1, \ldots, \iota_p \) and \( b_1, \ldots, b_p \) defined so that \( S_i = \{\iota_1, \ldots, \iota_p\} \), and each \( b_j \) is drawn from \( \{0, 1\} \) independently and uniformly at random. The hard-coded values \( \vec{a} = (a_0, \ldots, a_{\eta-1}) \) are defined as \( \vec{a} = \text{Sim}(b_1, \ldots, b_p) \).
3. \( \tilde{s} \leftarrow \text{GbMem}(K_{FA}, \text{OMem}(K_{ORAM}, s)) \).
Hybrid $H_{i,3}$: Hybrid $H_{i,3}$ is defined as the distribution on $(\tilde{M}, \tilde{s})$ obtained by sampling:

1. $K_{FA} \leftarrow \text{KeyGen}'(1^\lambda, T, S)$, $K_{ORAM} \leftarrow \text{Setup}(1^\lambda, S)$, and puncturable PRFs $F$ and $G$.

2. $\tilde{M} \leftarrow \text{GbPrg}(K_{FA}, M_{i,3})$, where the RAM machine $M_{i,3}$’s transition function is given in Algorithm 12, with the hard-coded values $\vec{a} = (a_0, \ldots, a_{\eta-1})$ sampled as $\vec{a} = \text{Sim}(b_1, \ldots, b_p)$ for uniformly random $(b_1, \ldots, b_p) \in \{0, 1\}^p$.

3. $\tilde{s} \leftarrow \text{GbMem}(K_{FA}, \text{OMem}(K_{ORAM}, s))$.

Data: Puncturable PRF $F$, punctured PRF $G' = G\{i\}$, underlying transition function $C'$,

$y = M(s)$, addresses $a_0, \ldots, a_{\eta-1}$, the running time $t^* = \text{Time}(M, s)$

Input: State $(t, q_{in})$, symbol $\sigma$

1. Let $(i', j')$ be integers such that $t = \eta i' + j'$ for $0 \leq j' < \eta$;
2. if $i' < i$ then
3. \hspace{1em} $(q_{out}, \text{op}) := C'^F(q_{in}, \sigma)$;
4. \hspace{1em} return $((t+1, q_{out}), \text{op})$;
5. else if $i' = i$ then return $((t+1, q_{in}), (a_j, \bot))$;
6. else if $i < i' < t^*$ then return $((t+1, q_{in}), (\text{Sim}(G(i'))_{j'}, \bot))$;
7. else return $y$;

Algorithm 12: Hybrid transition function $C_{i,3}$.

Hybrid $H_{i,4}$: Hybrid $H_{i,4}$ is defined as the distribution on $(\tilde{M}, \tilde{s})$ obtained by sampling:

1. $K_{FA} \leftarrow \text{KeyGen}'(1^\lambda, T, S)$, $K_{ORAM} \leftarrow \text{Setup}(1^\lambda, S)$, and puncturable PRFs $F$ and $G$.

2. $\tilde{M} \leftarrow \text{GbPrg}(K_{FA}, M_{i,4})$, where the RAM machine $M_{i,4}$’s transition function is given in Algorithm 12, with the hard-coded values $\vec{a} = (a_0, \ldots, a_{\eta-1})$ sampled as $\vec{a} = \text{Sim}(G(i))$.

3. $\tilde{s} \leftarrow \text{GbMem}(K_{FA}, \text{OMem}(K_{ORAM}, s))$.

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

Proof. This follows from fixed-access security.

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

Proof. This follows from the pseudorandomness of $F$ at the (selectively) punctured points $\{t_1, \ldots, t_p\}$ in a straight-forward way.

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

Proof. This follows from fixed-access security. Indeed, $t_1, \ldots, t_p$ are defined so that changing $F(t_j)$ can only possibly change the addresses accessed at time $t_x - i$. But at time $t_x - i$, the accessed addresses $a_0, \ldots, a_{\eta-1}$ are hard-coded with the same values in both $M_{i,2}$ and $M'_{i,3}$.

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

Proof. This follows from the pseudorandomness of $G$ at the selectively punctured point $t_x - i$ in a straight-forward way.

Claim 6.5.5. For all $i$ with $0 \leq i < t^*$, $H_{i,4} \approx H_{i+1}$

Proof. This follows from fixed-access security because $M'_{i,4}$ and $M'_{i+1}$ access the same sequence of addresses.
This concludes the proof of Lemma 6.4.

7 Persistent Data

The garbled RAM construction and security proof above can be generalized to a setting in which the garbled RAM programs act on a persistent database. That is, the updates that a garbled RAM program makes to a database \( D \) are accessible to the next garbled program to be executed on that database.

**Definition 7.1.** A RAM garbling scheme with persistent data is a tuple of p.p.t. algorithms \((\text{KeyGen}, \text{GbPrg}, \text{GbMem})\):

**KeyGeneration.** \( \text{KeyGen}(1^\lambda, T, S) \) takes as input a security parameter \( \lambda \) in unary, as well as time and space bounds \( T \) and \( S \), and outputs a secret key \( SK \).

**Program Garbling.** \( \text{GbPrg}(SK, M_i, i) \) takes as input a secret key \( SK \), a RAM machine \( M_i \) and an index \( i \), and outputs a RAM program \( \tilde{M}_i \).

**Memory Garbling.** \( \text{GbMem}(SK, s) \) takes a secret key \( SK \) and a memory configuration \( s \), and outputs a memory configuration database \( \tilde{s} \).

**Definition 7.2.** A RAM garbling scheme with persistent data is said to be **correct** if for every memory configuration \( s_0 \), for every \( \ell = \text{poly}(\lambda) \), and every tuple of RAM machines \((M_1, \ldots, M_\ell)\), it holds that the outputs of the garbled machines, when run in sequence on the garbled data, equal the outputs of the plaintext machines when run in sequence on the plaintext data. That is:

\[
\begin{align*}
\Pr \left[ y_1 = y'_1 \land \cdots \land y_\ell = y'_\ell \right] & \geq 1 - \text{negl}(\lambda) \\
S & \leftarrow \text{KeyGen}(1^\lambda) \\
\tilde{s}_0 & \leftarrow \text{GbMem}(SK, s_0) \\
\text{For } i = 1, \ldots, \ell \\
\tilde{M}_i & \leftarrow \text{GbPrg}(SK, M_i, i) \\
y_i & \leftarrow M_i(s_{i-1}) \\
y'_i & \leftarrow \tilde{M}_i(\tilde{s}_{i-1}) \\
s_i & \leftarrow \text{NextMem}(M_i, s_{i-1}) \\
\tilde{s}_i & \leftarrow \text{NextMem}(\tilde{M}_i, \tilde{s}_{i-1})
\end{align*}
\]

**Definition 7.3.** A RAM garbling scheme with persistent data is said to be **secure** if there is a p.p.t. algorithm \( \text{Sim} \) such that for all memory configurations \( s_0 \), all \( \ell = \text{poly}(\lambda) \), and all RAM machines \( M_1, \ldots, M_\ell \), no p.p.t. algorithm \( A \) can correctly distinguish between

\[
\begin{align*}
(\tilde{s}_0, \tilde{M}_0, \ldots, \tilde{M}_\ell) & \\
S & \leftarrow \text{KeyGen}(1^\lambda) \\
\tilde{s}_0 & \leftarrow \text{GbMem}(SK, s_0) \\
\text{For } i = 1, \ldots, \ell \\
\tilde{M}_i & \leftarrow \text{GbPrg}(SK, M_i, i)
\end{align*}
\]

and

\[
\begin{align*}
\text{Sim}(y_1, \ldots, y_\ell, s_0, |M_0|, \ldots, |M_\ell|, 1^\lambda) & \\
\text{For } i = 1, \ldots, \ell \\
y_i & \leftarrow M_i(s_{i-1}) \\
s_i & \leftarrow \text{NextMem}(M_i, s_{i-1})
\end{align*}
\]

with probability greater than \( \frac{1}{2} + \text{negl}(\lambda) \).

**Theorem 7.1.** If there is an indistinguishability obfuscator for circuits and there exist one-way functions, then there is a correct, secure RAM garbling scheme with persistent data.
Proof. (Sketch.) The scheme and the analysis are straightforward extensions of the single-machine case. That is, the memory garbling is identical to the single-machine case, except that “step 0” is attached to the root of the merkle tree before signing; The ith machine is garbled by applying the machine garbling algorithm of the single-machine case, with the exception that now the signature on the root of the Merkle tree is expected to contain also “step i − 1”, and the root of the Merkle-tree-hash of the final memory configuration is signed together with “step i”. (All machines are garbled with the same accumulator, iterator, and splittable signature parameters.)

Correctness, efficiency and succinctness are straightforward. For security, recall that the single-machine simulator generates a dummy (but legal) initial memory configuration, and a dummy machine that first verifies the signature on the memory configuration and then runs a dummy computation for a fixed number of steps at the end of which a hardcoded value is output. Here we extend this simulation strategy in the natural way. That is, the simulator first generates a dummy legal initial memory configuration. The ith dummy machine first checks the signature on its initial memory configuration, and verifies that the signed string has “step i − 1” encoded in it. Then the machine runs a dummy computation for a fixed number of steps, outputs the hardcoded value, and signs the final memory configuration along with “step i”. Analysis of the simulator is extended in a natural way. In particular, it can be done in the same modular way as in the single-machine case.

We note that our notion of garbling with persistent data generalizes garbling with a long output. Recent work [LPST15] has shown that full “compactness” is impossible in this setting. Indeed, the combined size of \( M_1, \ldots, M_\ell \) in our scheme grows proportionally to the total output length.

Acknowledgements

This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467.

References


A ORAM Construction

Memory Layout  Recall that for a RAM machine $M$ and a (dense) memory configuration $s$, the Chung-Pass ORAM begins by partitioning $s$ into blocks of size $\alpha$ for some constant $\alpha > 1$. These blocks are labeled with their addresses, randomly shuffled, and a position map $\text{Pos}_1$ is generated, with $\text{Pos}_i(i)$ storing the shuffled position of block $i$. For $i = 1, \ldots, \log(\|s\|_0)$, $\text{Pos}_i$ is also randomly shuffled in blocks of size $\alpha$, with $\text{Pos}_{i+1}$ storing this shuffling.

For each $i$, there is a balanced binary tree $T_i$ of buckets in memory, with each bucket sized to hold $\text{polylog}(\lambda)$ blocks. When $\text{pos}_i(x) = y$, it means that the block labeled $x$ is present in this tree in one of the buckets from the root to leaf $y$.

Memory Accesses  To obliviously access an address $a$ in this data structure, one first looks up $\text{Pos}_{\|s\|_0}(a)$ from private registers or a brute-force ORAM. Having computed $\text{Pos}_i(a)$, one now looks up $\text{Pos}_{i-1}(a)$ by reading each bucket on the path to the $\text{Pos}_i(a)^{th}$ leaf in $T_i$, searching for the block which is labeled $a$ (or more precisely $\lfloor a/\alpha^i \rfloor$) and retrieving its contents. This block is not written back to the bucket, but is instead assigned a new position $\text{pos}^*$. Labeled as such, it is inserted into the root bucket in $T_i$. Next, and this step is crucial to prevent buckets from overflowing, the path in $T_i$ from root to a random leaf is traversed, with each block moved to the leafmost admissible bucket along this path.

Remark 1. While the Chung-Pass ORAM is defined for dense initial memory configurations, one can apply it to sparse memory configurations with only polylogarithmic overhead. For example, one can densely encode a balanced binary tree representing the sparse memory configuration, and then apply the ORAM to this dense encoding.

However, we allow the transformed memory configuration to be sparse – this enables the space bound $S$ to be larger than the initial memory size $\|s_0\|$. In particular, a shuffling of blocks of $\|s_0\|$ in the address space $[S]$ can be represented with size $O(\|s_0\| \log S)$. This new sparseness feature, introduced in the name of efficiency, also introduces a new security requirement: The sparsity pattern (the non-empty addresses) of the transformed initial memory configuration must be independent of the plaintext memory configuration’s contents (but not length).

A.1 Localized Randomness

In order to assert the localized randomness property, we must define an algorithm $\text{Sim}$, and given a RAM machine $M$ and memory configuration $s$ with $\text{Time}(M,s) = t$, we must define subsets $S_1, \ldots, S_t$ of the ORAM randomness such that the addresses accessed on the $i^{th}$ access when using randomness $\vec{r}$ are given by $\text{Sim}(\vec{r}_{S_i})$.

$\text{Sim}$ takes a random string $r$ and interprets it as labeling two leaf nodes in each tree $T_{\|s\|_0}, \ldots, T_1$. For each such leaf node $\ell$, $\text{Sim}$ outputs the addresses of each bucket on the path from the root to $\ell$.

Suppose that $M$ on $s$ accesses addresses $a_1, \ldots, a_t$. The set $S_i$ is defined by considering, for each $j \in \{1, \ldots, \|s\|_0\}$, $k_{i,j} = \max\{k : k < i \land a_k/\alpha^j = a_i/\alpha^j\}$. Then $S_i$ is defined as the concatenation of, for each $j$, the randomness used in choosing the new positions $\text{pos}^*$ in $T_j$ on the $k_{i,j}^{th}$ underlying access, as well as the randomness in choosing the random path along which blocks should be flushed towards the root.